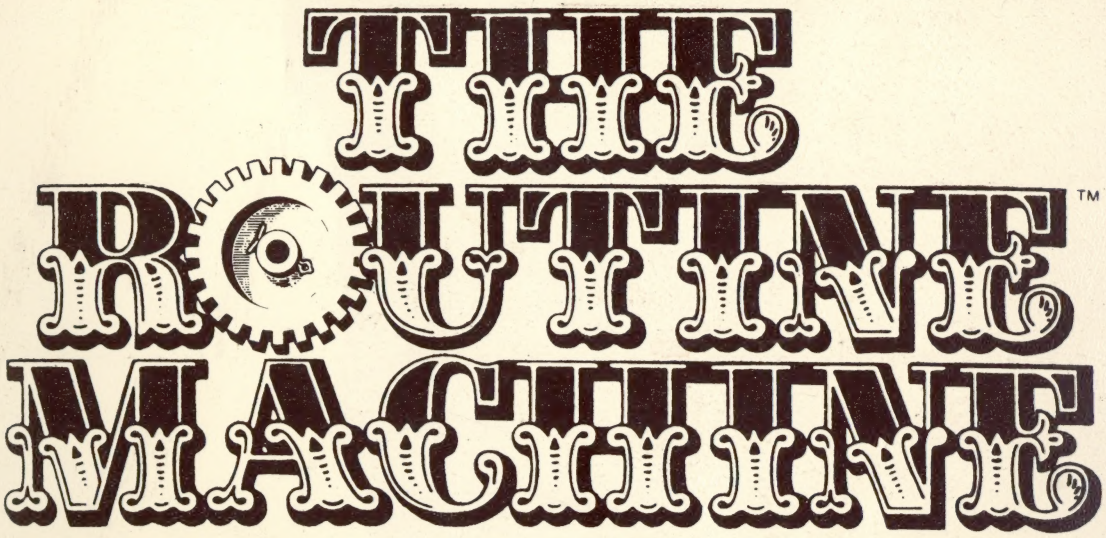


A Versatile Programming Utility
for the Apple II

SDS southwestern data systems™

THE ROUTINE MACHINE™

by Peter Meyer

PROGRAM CONSULTANTS:
Craig Peterson
Glen Bredon
and
Roger Wagner

INSTRUCTION MANUAL

PRODUCED BY:

SDS™

southwestern data systems™

10761-E Woodside Avenue • Santee, California 92071
Telephone: 619/562-3670

SOUTHWESTERN DATA SYSTEMS
CUSTOMER LICENSE AGREEMENT

IMPORTANT: The Southwestern Data Systems software product that you have just received from Southwestern Data Systems, or one of its authorized dealers, is provided to you subject to the Terms and Conditions of this Software License Agreement.

Should you decide that you cannot accept these Terms and Conditions, then you must return your product with all documentation and this License marked "REFUSED" within the 30 day examination period following the receipt of the product.

1. License. Southwestern Data Systems hereby grants you, upon your receipt of this product, a nonexclusive license to use the enclosed Southwestern Data Systems product subject to the terms and restrictions set forth in this License Agreement.

2. Copyright. This software product, and its documentation, is copyrighted by Southwestern Data Systems. You may not copy or otherwise reproduce the product or any part of it except as expressly permitted in this License.

3. Restrictions on Use and Transfer. The original and any back-up copies of this product are intended for your personal use in connection with a single computer. You may not distribute copies of, or any part of, this product without the express written permission of Southwestern Data Systems.

DISCLAIMER

SOUTHWESTERN DATA SYSTEMS AND THE PROGRAM AUTHOR SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO PURCHASER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY THIS SOFTWARE, INCLUDING, BUT NOT LIMITED TO ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THIS SOFTWARE.

COMMERCIAL USE OF SOFTWARE

The Routine Machine software package is licensed for your use within programs created for your own personal use. If you intend to commercially distribute programs which contain the Routine Machine Interface Routine and/or additional Library Routines, please contact Southwestern Data Systems for commercial licensing terms.

*** THE ROUTINE MACHINE ***

An assembly language program by Peter Meyer

Documentation by Roger Wagner and Peter Meyer

APPENDICES (Continued)

LIBRARY MODULES

LIBRARY MODULES (Continued)

INTRODUCTION

The Routine Machine is a versatile programming utility whose main purpose is to allow you to incorporate machine language subroutines within your Applesoft programs, and to pass variables back and forth between the main Applesoft program and its machine language subroutines as easily as is done within Applesoft itself. In this way programs can be created which combine the speed and power of machine language with the simplicity and convenience of Applesoft BASIC.

Much more than just a library of miscellaneous routines, the Routine Machine and its companion Ampersoft Program Library disks actually EXTEND Applesoft to a degree which approaches an entirely new programming language. The Routine Machine allows you to write Applesoft programs in a new and remarkably efficient way, in that you can now construct your programs using pre-written modules which accomplish a specific purpose. Each module is invoked using a one-line command, which is far simpler than employing (to say nothing of writing) a complex Applesoft subroutine to do the same task.

The usual problems with putting machine language code in a program are:

1. Required knowledge of machine language programming.
2. A way of interfacing the routine to Applesoft, particularly the passing of variables back and forth.
3. Deciding where in memory to locate the routines.
4. Avoiding conflicts when using more than one routine.
5. Finding (or worse, writing and debugging) the routines you really need.

The Routine Machine SOLVES ALL THESE PROBLEMS in one easy-to-use package. The Routine Machine does not require knowledge of machine language programming, and makes the routines accessible by allowing you to choose from a professionally written library. Routines are invoked from your Applesoft program using names chosen by you, and there is never any need to use BLOADs or to concern yourself either with the locations of the routines in memory or with their protection during the running of the program.

The Routine Machine diskette includes thirty one valuable routines (as listed in the Table of Contents). Additional routines are available on the continuing series of Ampersoft Program Library diskettes. The Routine Machine can also be used with other machine language subroutines which are either written by yourself or obtained from another source such as the computer magazines.

YOUR DISKETTE

You do not have to boot on the Routine Machine diskette to use any of the programs on the diskette. If the power was not previously on, or if you wish to install the modified DOS used on the Routine Machine diskette, you may however boot in the usual manner. This will not affect any programs you may wish to run later.

IMPORTANT: It is strongly advised that you do not use your originally purchased diskette, in either the examples that follow, or your daily use of Routine Machine. Instead, make a back-up as described in Appendix A of this manual. The original should then be put in a safe place, and the back-up used as your work diskette. After making your work diskette, return to this section.

If you have booted on the Routine Machine diskette, typing in CATALOG should give the following display:

```
DISK VOLUME 254                    281

*A 019 HELLO

THE ROUTINE MACHINE
-------------------
*B 003 ROUTINE MACHINE
*B 032 NATTY DREAD

*T 002 AMPERSAND SETUP
*T 002 CALL SETUP
*A 006 JUMP FILE CREATE
*B 003 REMOVE ROUTINE MACHINE

LIBRARY FILES
-------------
*B 002 SWAP.RM
*B 003 PRINT USING.RM
*B 002 TEXT OUTPUT.RM
*B 002 STRING INPUT.RM

*B 002 SEARCH.RM
*B 003 ARRAY SEARCH.RM
*B 002 BUBBLE SORT.RM
```

```
*B 002 BEEP.RM
*B 002 SOUND EFFECTS.RM

*B 002 ERR.RM
*B 002 ERR MSSG.RM

*B 002 GOTO.RM
*B 002 GOSUB.RM
*B 002 LINE DATA RESTORE.RM
*B 002 DATA ELEMENT SELECT.RM
*B 002 XNUM.RM
*B 003 MEMORY MOVE.RM
*B 002 RESTORE AMPERSAND.RM
*B 002 PTR READ.RM
*B 002 PTR WRITE.RM

*A 013 SHAPE GOBBLER
*A 016 SHAPE TABLE VIEWER
*B 006 HIRES ASCII.RM
*B 004 TURTLE GRAPHICS.RM
*B 005 TURTLE GRAPHICS+.RM

*B 004 BLOAD.RM
*B 003 BINADR.RM

*B 002 RESET NORM.RM
*B 002 RESET ONERR.RM
*B 002 RESET RUN.RM
*B 002 RESET BOOT.RM

*B 002 FREE SECTOR COUNT.RM
*B 002 FIX LINK FIELDS.RM
```

The programs on this diskette have been organized for ease of use, and are appropriately divided by their application. The Routine Machine consists of two machine language programs:

1) ROUTINE MACHINE
2) NATTY DREAD

The diskette also contains a program to remove the ROUTINE MACHINE (called REMOVE ROUTINE MACHINE) and two files (AMPERSAND SETUP and CALL SETUP) which are used to create a basic hook-up line in any Applesoft program employing Routine Machine modules. JUMP FILE CREATE is a utility provided to link blocks of non-relocatable routines to your Routine Machine programs. (See Appendix C, on advanced techniques, for more details on this.)

The remaining files are routines with which to begin your library of machine language routines for use with the Routine Machine. A few are mentioned within the main text of this manual. All library modules are described in detail in the special section at the end of this manual.

In addition the backside of the Routine Machine diskette contains a number of demonstration programs. The programs are not intended to provide complete explanations of how to use each routine, but will provide some insight as to the possible uses.

The side of the diskette with the demonstration programs is not protected in any way, and may be copied using any standard disk copy program such as COPYA on your Apple System Master Diskette.

SPECIAL DOS FEATURES

If you do boot on the Routine Machine diskette, a special Disk Operating System (DOS) will be in effect which you may find to be of some advantage.

The first thing you will notice is a number just to the right of the 'DISK VOLUME 254' message, similar to the '281' in the sample directory listed on the previous page. This is the number of free sectors remaining on the diskette being CATALOGed. A normal Apple DOS 3.3 diskette has a maximum of 496 free sectors available.

The second feature is an optional termination of the CATALOG listing at any of the pauses which usually occur when a directory has more files than can be displayed on the screen at one time. When using the modified DOS, pressing RETURN will terminate the CATALOG listing at any pause. Pressing any other key will continue it.

Although not a specific function of the Routine Machine DOS, it should be mentioned that whenever the Routine Machine diskette is booted, two options are possible.

If the 'C' key is pressed during the boot process, the back-up copy program will run.

If the 'C' key is not pressed, then you will be asked whether you wish to run Routine Machine or not. If you answer 'No', the diskette CATALOG will be displayed, and the user left with the usual Applesoft prompt (']').

The 'free space' and 'catalog clip' features may be propagated to other diskettes by INITing them after first booting on the Routine Machine diskette. It is not necessary to use the copy program just to propagate these features.

A SHORT DEMONSTRATION

The primary purpose of Routine Machine is to make pre-written machine language routines available to your own Applesoft programs, as easily and efficiently as possible. All that you have to do is:

1) Insert a line in your program which sets up the ampersand vector to connect your Applesoft program with the routines appended.

2) Use the Routine Machine to append the routines of your choice.

3) Invoke these routines from within your Applesoft program by using the ampersand followed by the name of the appended routine.

To see how this works, let's consider an actual example. Let's suppose for a moment that you would like to have some kind of simple tone routine available to a program you are writing. The entire process can be done in a matter of minutes:

1. Type in 'FP' to assure a clean entry into Applesoft BASIC.

2. With the Routine Machine disk in the drive, type in EXEC AMPERSAND SETUP. Then type in LIST and you should see:

   ```
   1 CALL PEEK(175) + 256 * PEEK(176) - 46
   ```

   When your program is run this line will establish a connection (by means of the ampersand) to the various library modules you have appended.

3. Get the Routine Machine up and running by typing in:

   ```
   BRUN ROUTINE MACHINE
   ```

The routine ROUTINE MACHINE is actually a loader program which moves any Applesoft program in memory so that it will not be overwritten, and then installs the main Routine Machine utility program, NATTY DREAD. After a few seconds the main menu will appear:

```
                *** ROUTINE MACHINE ***

          SELECT AN OPTION:

     1.  APPEND A MODULE
     2.  REMOVE A MODULE
     3.  REMOVE ALL MODULES
     4.  COPY ALL APPENDED CODE TO DISK
     5.  RESTORE APPENDED CODE FROM DISK
     6.  REPORT MODULES APPENDED
     7.  SEARCH FOR AMPERSANDS/CALLS
     8.  INSPECT APPLESOFT LINE
     9.  DISPLAY MEMORY MAP
     0.  EXIT
                                [NO MODULES APPENDED]
```

4. Since the first thing we want to do is to append a new routine to our program, press '1'. The screen will then display:

```
INSERT DISK CONTAINING SUBROUTINE TO BE APPENDED,
THEN ENTER NAME OF FILE

('CAT' FOR CATALOG, <RETURN> TO QUIT)

FILENAME ->
```

If you knew that BEEP.RM was the file you wanted to append at this point, you could just enter the name directly. Often you will not know the name exactly, in which case you can enter 'CAT' to produce a catalog of the diskette. Enter 'CAT' now. Since BEEP.RM is further down the list, press the space bar (not RETURN) for the next page of the catalog. BEEP.RM should now be visible. Now press RETURN to stop the catalog (assuming that you have booted with the Routine Machine diskette).

5. Now type in ´BEEP.RM´ as the library module to be loaded. Note that all Routine Machine module names have a ´.RM´ suffix.

   After specifying the routine, the screen will display:

   INVOCATION NAME ->

6. You must now enter the name (or character string) by which you will call this particular routine from within your program. Because the library modules themselves may have rather long names (so as to be most descriptive) you will often want to use a shorter name when invoking the routine from within your program. In this case, ´TONE´ will do nicely. Type in ´TONE´ as the name and press RETURN.

The Routine Machine will load the BEEP.RM routine, and give it the name ´TONE´ within your program. Different names may be used within different programs, although this is not recommended.

After the routine is appended, the program will return to the request for a new FILENAME. This is because usually you will want to append several routines at one time. Press RETURN alone to return to the menu at this point.

7. Now enter ´0´ and press RETURN to exit the Routine Machine and return to your program.

8. After (or, if you wish, before) appending a module to your Applesoft program you can write lines in your program which will invoke that module and cause it to perform its designated task. Such a line is known as a ´module invocation´, and has either two or three parts, as follows:

The first part required is the ampersand symbol (&). You can think of this as being similar to the Control-D character needed whenever a disk command is done. The difference here is that the ampersand will invoke a Routine Machine appended module.

The second item required is the invocation name (in quotes) for the routine you wish to call. For our example, this would be "TONE". If the routine which you have just appended does not require any information to be passed to it, then nothing more is required in the module invocation.

The third item is optional, an is usually referred to as a "parameter list". Each Routine Machine library module may or may not have certain pieces of information, called PARAMETERS, which have to be passed to it. In the case of the BEEP.RM routine, these parameters are pitch and duration. (In the case of other modules, the parameter list following the name may have a different number of parameters required and use a variety of variable types.)

To use this in a program, you would first determine the values for pitch (P) and duration (D), such as through an INPUT statement, and then call the routine via the ampersand statement. To show how this might be done, enter the following program lines (these will now be in addition to the line #1 which was previously entered via the AMPERSAND SETUP procedure above):

```
10 INPUT "ENTER PITCH, DURATION: "; P, D
20 & "TONE", P, D
30 PRINT: GOTO 10
```

9. Now just RUN the program to try it out! Enter:

```
10,10
50,10
100,20
```

as some sample number pairs. Notice that duration is fairly linear in that a duration of 200 will produce a tone twice as long as a duration of 100. The duration is also measured in 100ths of a second; thus a value of '100' gives a tone about 1 second long.

YOU'RE DONE! You now have a complete program. The program can be LOADed or SAVEd just like any normal Applesoft program, and the machine language routines will be carried along automatically. They will remain a permanent part of your Applesoft program, unless you decide later to remove them using the Routine Machine's "Remove a Module" option. If you want to add more routines later, just BRUN ROUTINE MACHINE again, and add the modules you want. See the section at the end of this manual to determine the exact syntax for invoking each module.

ADDING MORE ROUTINES / RESUMING NORMAL OPERATION

If you want to add a new module right away, type in CALL 2051 to re-enter the Routine Machine. If you're done using the Routine Machine for awhile and want to free up the memory normally occupied by the main Routine Machine utility, type in: BRUN REMOVE ROUTINE MACHINE. This will remove the main Routine Machine program from memory and will then relocate your Applesoft program back to its usual location ($801 = 2049).

The Routine Machine program is required in memory only when you are appending or removing modules (or doing any of the other things that it allows you to do). It is not required to be in memory when you run your program.

After you have appended a module with the Routine Machine, you may exit and immediately run your program (with the Routine Machine still present). Alternatively, you may BRUN REMOVE ROUTINE MACHINE as described above and then run your program at its usual location. This is recommended in the case of very large programs or programs that employ graphics. (See the section "A Little More Detail" for more on memory usage by the Routine Machine.) Another way to put your program back where it belongs is first to SAVE it on disk, enter 'FP' and then LOAD your program. (The 'FP' command does not in itself wipe out the Routine Machine. If you accidentally do an 'FP' with the Routine Machine installed you can recover simply by re-entering the Routine Machine with a CALL 2051.)

You've now learned most of what it takes to use and enjoy the Routine Machine. You could now skip to the library section to see what routines have been included in this package. Their wide variety makes the Routine Machine of immediate value in your programming efforts, no matter what area you may be working in.

If you are impatient and wish to get your hands on the Routine Machine right away then go ahead - it's quite friendly. At some time, however, this documentation should be read, as it does provide much useful information on how to get the most out of this unique programming tool.

IF IT DOESN´T WORK

In order for your program to invoke an appended module two conditions must be fulfilled: The ampersand link must have been set up before any module invocation is executed in your program, and there must actually be an appended module with the name used in your module invocation. If either of these conditions is not met, your program will crash unceremoniously.

THE AMPERSAND LINK. It is good practice, when developing a program which will invoke Routine Machine modules, to BEGIN by EXECing the AMPERSAND SETUP file. As explained earlier, this inserts a line (with line #1) in your program which points the ampersand vector to the right place (actually, to the ´Interface Routine´, which handles all your module invocations). This line can occur anywhere in your program, provided that it is executed prior to any module invocation. If it is not present then the probable consequence of attempting to invoke a module will be a SYNTAX ERROR. If this happens when you run your program, check if the ampersand setup line is present AND being executed prior to the module invocation.

THE APPENDED MODULE. Another way to bomb your program is to attempt to invoke a module which has not been appended. In this case (assuming that the proper setup line has been inserted) the Interface Routine will try to find the module named by you among the modules appended. When it doesn´t find it, it will generate an UNDEFINED FUNCTION error message (with the line number). You can then ascertain which module is missing.

If you are sure that both the ampersand link and appended module are present, then double-check the syntax (and perhaps the sample program listing) for the routine as listed in the section on the LIBRARY MODULES at the back of this manual.

IMPORTANT NOTE: The Routine Machine may be used to append ANY machine language routine to an Applesoft program, and is not limited to the routines available on the Routine Machine disk and the Ampersoft Program Library disks. However, if you wish to append a routine from a source other than these library disks, the routine MUST first be placed on a legitimate copy of the Routine Machine or an Ampersoft Program Library diskette. (This is easily done using the Binary File Copier program which will be found on the backside of the Routine Machine disk.)

A LITTLE MORE DETAIL

Now that you have an idea of what Routine Machine does, here are some additional details of what actually goes on to establish the connection between the desired routine and your Applesoft program. Although it is not necessary to understand all the information provided here to use Routine Machine, it may help you get a better feeling for what you are actually doing when you use the program.

When you first specify the name of a module to be put in your Applesoft program, Routine Machine does the following things:

1) It appends the routine to the end of your Applesoft program. Because of the nature of Applesoft, this code is not visible during a LIST, and is unaffected by adding or deleted normal BASIC lines or statements.

2) It saves both the name you give to invoke the routine, and the name of the disk file loaded. This is to make later identification of routines easier (see the section on the Module Report).

3) The first time a module is appended, Routine Machine adds its own 'Interface Routine' to the end of your program. The function of this is to locate a module named in a module invocation and to pass control to it.

The manner in which memory is normally allocated in the Apple II Plus is as follows:

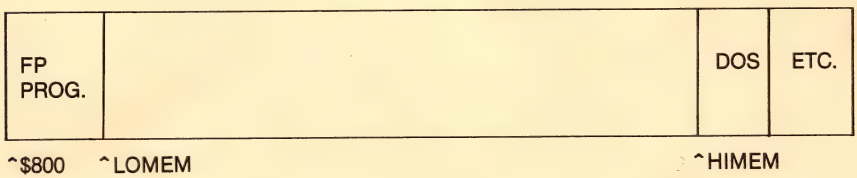


Figure 1: "Normal" Applesoft program

Normally an Applesoft program resides at the "bottom" of memory, with all remaining space above it reserved by DOS and for variable storage.

Routine Machine could be loaded right after your program, but as routines were added to the end of the Applesoft program, they would start to conflict with Routine Machine itself. To avoid this, Routine Machine moves your program UP in memory, and locates itself at memory location $800 (the dollar sign indicates a hexadecimal address).

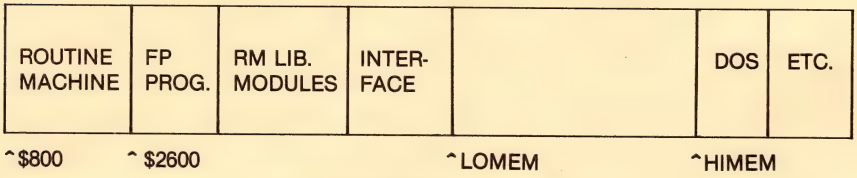


Figure 2: With Routine Machine loaded

With your program safely relocated to $2600, routines are automatically placed between the Interface Routine and the end of your program ('FP PROG' on the chart). When your program is run, the Line #1 that was put in by the AMPERSAND SETUP file points the ampersand vector to the Interface Routine at the end of your program. From there the name following the ampersand is read and the proper routine invoked.

CAUTION: Note that your program now STARTS at $2600. This is well into the Hi-Res page 1 display, which normally starts at $2000. Thus, any program using an HGR command will destroy itself if you attempt to run it with ROUTINE MACHINE itself in memory. Routine Machine should be removed before running any program using Hi-Res graphics.

Figure 3: At 'Run Time'

If you use the REMOVE ROUTINE MACHINE utility, or do a fresh load of an Applesoft program containing appended modules after entering ´FP´, then memory will be allocated as in Figure 3. This shows the Applesoft program back at its normal location, with the appended routines and the Interface Routine still present.

Hi-Res graphics on page 1 will in most cases now be available. When in doubt, you can use the Memory Map option from the Routine Machine (see p.38 of this manual) showing the addresses for your program at its normal location of $800. You can also use the PTR READ routine described in the library section to perform this check (end-of-program point greater than $2000) automatically.

When developing a program using Routine Machine modules, you can proceed in three ways:

1) Add modules to your program, one at a time, as required.

2) Work on the BASIC portion of your program first, and then add the required modules later, or

3) Append all required modules and then write the BASIC portion of the program which calls them.

DEMONSTRATION PROGRAMS

There are a number of demonstration programs in the Routine Machine package. These demos were created using Routine Machine to append machine language routines which were taken from among the library routines on the disk. These demo programs may be LISTed and studied to see exactly how the appended routines are used within an Applesoft program. (See p.162 of this manual for more on the demo programs.)

PARAMETER PASSING

The Routine Machine takes care of the details of appending a machine language routine, as well as making provisions for the connection between the routine and your Applesoft program. There is however, still the question of the syntax of the statement used in the Applesoft program to invoke the routine.

Each appended routine is invoked by means of the ampersand followed by the name of the routine in quotes, then any parameters required by the routine.

For example, we may have a routine which will sort the elements of a string array from one element, "I", to another element, "J". If, when appending the routine, we have given it the name 'SORT', then the calling statement could be:

```
& "SORT", A$(I) TO A$(J)
^  ^^^^   ^  ^        ^
1  2      3  4        5
```

where:

1. & calls up the Interface Routine
2. SORT is the invocation name for the module
3. A$() is name of the array to be sorted
4. I is the initial element in the sort range
5. J is the final element in the sort range.

What happens: When control eventually passes to the sort routine it will read the parameter list to find which array is to be operated on, and what range of elements in that array is to be sorted. After performing the sort operation, control is returned to the Applesoft program and then execution will continue with the next statement in the program.

The nature of the parameter list depends ENTIRELY on the routine being called. In the case of a string input routine, only the name of the string variable to be used is required. Thus, the statement used there would look like:

```
& "GET STRING", B$(I)
```

Some routines do not require any parameters to be passed. For example, there is a routine which will cause a program to re-RUN whenever RESET is pressed. This routine is provided on the Routine Machine diskette under the name RESET RUN.RM. To set up the RESET vector using the routine, you need only use a statement such as the following:

```
& "RESET RUN"
```

Some routines have OPTIONAL parameters. That is, you may include certain parameters in the parameter list or not, depending on just what you want the routine to do. For example, there is a fast binary file loader routine called BLOAD.RM on the diskette. This will load binary files such as Hi-Res pictures approximately five times faster than normal. The simplest invocation would be:

```
& "BLOAD", A$
```

where A$ (or any legal string expression) specified the name of the file to be BLOADed. As with the normal Applesoft BLOAD however, it is sometimes useful to be able to specify a distinct load address. This could be done by adding an optional address parameter:

```
& "BLOAD", A$, AD
```

where AD (or any legal numeric expression) specifies the address at which the file is to be BLOADed.

In most cases, commas are used to separate the variables in a parameter list in a Routine Machine module invocation. Where possible, we have also endeavoured to ensure that standard Applesoft conventions are observed. That is to say, if a string is needed, then any legitimate string expression, whether variable, string literal or string expression will be acceptable, just as it would be in Applesoft. For example, for the BLOAD just mentioned, all the following statements would be acceptable:

```
& "BLOAD", A$, AD

& "BLOAD", "FILENAME", 8192

& "BLOAD", A$ + ".PIC", AD + 192
```

It should also be noted that there are some special cases where variables are passed BACK from the routine to the calling Applesoft program. For example, the error-handling routine ERR.RM will return the error code and the error line in the variables EC and EL (or any other real variables) if these are included in the parameter list.

In fact, three forms of the module invocation are possible in this case:

```
& "ERR"            (fixes the stack only)

& "ERR", EC        (returns the error code in the
                    variable 'EC')

& "ERR", EC, EL    (also returns the line number of
                    the error in the variable 'EL')
```

* *

The following sections provide greater detail on each of the individual Routine Machine menu options. Information on the library routines on the Routine Machine diskette will be found at the end of this manual.

* *

ROUTINE MACHINE

>> ROUTINE MACHINE MAIN MENU <<

The following pages contain further details on each of the menu choices presented when running the main Routine Machine utility program. They are provided to answer any specific questions which might arise about a given menu choice.

MENU OPTION PREREQUISITES

For each of the options listed in the menu, certain conditions must be satisfied before the option can be selected. The prerequisites for each of the modules are stated below:

1	APPEND A MODULE	There must be a program in memory.
2	REMOVE A MODULE	At least one module must be appended.
3	REMOVE ALL MODULES	At least one module must be appended.
3*	REMOVE APPENDED MACHINE CODE	This will appear in the menu ONLY if there is foreign machine code appended, but no modules.
4	COPY ALL APPENDED CODE TO DISK	There must be appended code present, either foreign machine code, or appended modules.
5	RESTORE APPENDED CODE FROM DISK	This option can be exercised ONLY if no module is present. It can be used with foreign machine code present.
6	REPORT MODULES APPENDED	At least one module must be present.
7	SEARCH FOR CALLS AND AMPERSANDS	There must be a program in memory.
8	INSPECT APPLESOFT LINE	There must be a program in memory.
9	DISPLAY MEMORY MAP	There are no prerequisites.
10	EXIT	There are no prerequisites.

OPTION #1

APPENDING MODULES

Although most of the important points regarding this operation have already been covered earlier in this manual, let's briefly review the process, and fill in some of the finer points not mentioned previously.

Having developed, or partially developed, an Applesoft program in which you wish to incorporate certain library routines, you would carry out the following steps to make the subroutine a permanent part of your program.

1) If Routine Machine is already in memory, enter CALL 2051. If Routine Machine is not in memory, place the Routine Machine diskette in the drive and enter: BRUN ROUTINE MACHINE.

2) When the menu appears, select option 1 (APPEND A MODULE) by pressing '1'. The screen will then clear and the following text will be displayed:

> INSERT DISK CONTAINING SUBROUTINE TO BE APPENDED, THEN ENTER NAME OF FILE
>
> ('CAT' FOR CATALOG, <RETURN> TO QUIT)
>
> FILENAME ->

At this point you may insert either the Routine Machine diskette with its library files, or an Ampersoft Program Library diskette. These are additional library diskettes available from Southwestern Data Systems.

> IMPORTANT: Although you can use your own ampersand routines, such as those you've written yourself or derived from magazines, the routines must be placed on a Routine Machine or Ampersoft Program Library diskette to be usable with the Routine Machine.

3) If you know the name of the routine you wish to append, simply enter it. If you are unsure of the name, enter 'CAT' to catalog the diskette.

When you've found the file you want, type in the name, exactly as it appears in the catalog. Remember to use the special 'catalog clip' feature (press RETURN at a pause) to keep the name you want on the screen.

4) The screen will then display:

INVOCATION NAME ->

The name used to invoke the routine can be anything you like, subject to the following restrictions:

a) The maximum length of the invocation name is fifteen characters.
b) Control characters are not permitted; otherwise all keyboard characters other than ´]´ and quote marks (") are permitted.
c) The first character must not be a space; otherwise spaces within the name are permitted.

The name used to invoke a module is also independent of the name of the disk file containing the routine.

When you enter the name you chose to invoke the routine, Routine Machine checks to see whether this name has been used for any previously appended module. When calling the module from your Applesoft program, the name specified after the ampersand must, of course, be exactly the same as the name used when the module was appended to your program. If not, you will get an UNDEFINED FUNCTION error.

If you should forget the name you gave a particular routine, you can always use Option #6 (Module Report) to get a report on the invocation names (and the original file names) of all currently appended modules.

5) Once the file name and invocation name have been entered, the Routine Machine:

a) Searches for that file name and if found,
b) appends the routine at the end of your Applesoft program, along with the new module name, then
c) adjusts the end-of-program pointer for your original program as necessary.

If the disk file that you have specified is not on the disk, Routine Machine will display a FILE NOT FOUND error message and again ask you what file name you wish to append.

REMOVING APPENDED MODULES

OPTION 2: REMOVE A MODULE

Suppose you have appended 23 modules and then decide you really don't want module #5. Routine Machine allows you to remove it easily. Select '2' at the menu, and the first page of a module report will appear. This chart will show all of the appended modules, giving both the invocation name, and the name of the file from which the original binary routine was loaded.

The modules are shown eight at a time. If you want to go to the next eight, simply press the space bar to advance. You can also return to the main menu at any point by pressing RETURN alone.

If you see the module you wish to remove, press the slash key ('/') to tell the program you wish to remove an item. Then enter the number of the module you wish to remove.

Routine Machine will then remove the specified module and return you to the main menu. The other modules will not be affected and if you request a Module Report (see Option #6), you will find all the modules intact, except for the one removed.

OPTION 3: REMOVE ALL MODULES

It is also possible to remove all appended modules at one time. Select option #3 for this function. You will be asked to confirm your intentions for such a drastic action. You must respond with either 'Y' or 'N' to this question.

After removing all modules, if you ask for a Module Report (option #6), the program will beep to signify no modules present. Additionally, you may notice that next to the EXIT option NO MODULES APPENDED is displayed in inverse.

SPECIAL NOTES:

It is possible to append Routine Machine modules to an Applesoft program that already has machine code appended ('foreign machine code'). Normally this will not affect the functioning of this previously-appended machine code. However the converse case is not possible: The Interface Routine must be at the end of the appended machine code, and so foreign machine code should not be appended to a program with Routine Machine modules.

In the case where your Applesoft program has foreign machine code appended, Option 3 changes slightly. Consider, first, the case of an Applesoft program which has both foreign machine code (following the BASIC portion) and one or more appended modules. In this instance, Option 3 of the menu will read REMOVE ALL MODULES. If you use this option, Routine Machine will remove all appended modules and the Interface Routine, but leave the foreign machine code intact.

On return to the menu, Option 3 will then read REMOVE ALL APPENDED MACHINE CODE. If you now select Option 3, the foreign code will then be removed, leaving your program with no code appended whatsoever. This duality of Option 3 does not occur if the machine code appended to the end of your program consists only of modules which have been appended using Routine Machine.

OPTION #4

COPYING ALL MODULES TO DISK

After using Routine Machine for awhile, you will probably find that there is a certain "core" of routines which you almost always put in your programs. Option 4 may be used for saving an assembled group of routines to disk as a single unit. This enables you to append the entire group of routines at one time (using Option 5) to an Applesoft program in the course of development.

Such a mini-library can really come in handy when you write a variety of different programs. An example of a group of routines which could make up a good mini-library include:

a) STRING INPUT.RM
b) PRINT USING.RM
c) TEXT OUTPUT.RM
d) ERR.RM
e) BEEP.RM

It is also useful to make up packages of special-application routines concerned, for example, with string array manipulation or real variable operations.

The second main application of this option is to save appended modules to disk to enable the performance of some operation which might damage appended machine code. Although none of Southwestern Data Systems' software will damage appended machine code, the same cannot be said for all utilities. In particular, Apple Computer's Renumber program WIPES OUT all machine code appended to an Applesoft program.

It is not difficult to use Apple Computer's Renumber program to renumber an Applesoft program which has Routine Machine modules appended, but the proper procedure must be observed. The following is the method which must be employed.

Suppose that you have a partially developed Applesoft program to which you have appended a number of Routine Machine modules, and that you wish to renumber your program. Run Apple's Renumber program to install the renumber utility. BRUN ROUTINE MACHINE, then load your program but do NOT run it. CALL 2051 to enter the Routine Machine. Select Option 4 (press RETURN to confirm your selection). The following display will then appear:

```
ALL APPENDED MACHINE CODE WILL NOW
BE COPIED TO A FILE ON DISK

DO YOU WISH TO USE 'A.M.C.'
AS THE FILE NAME?  [Y]
```

The Routine Machine is now ready to take all the machine code which is appended to your program and place it in a binary file on the disk. In this case you don't care what the file is called, so you can accept the default name 'A.M.C' ('Appended Machine Code'). (If, however, you were saving a package of modules for later use then you would provide a name of your own for the file.)

After you have provided the filename, the Routine Machine will save the file. The modules will still be appended, but there is no need to remove them. Now exit the Routine Machine (Option 0) and renumber your program in the usual way. Since you did not run your program, the ampersand vector will still be pointing to the renumber routine (which is located just above HIMEM).

Having renumbered to your satisfaction, CALL 2051 to re-enter the Routine Machine, and select Option 5, RESTORE ALL APPENDED CODE FROM DISK, as described in the next section. Your previously appended modules will now be restored as they were before. You can now exit the Routine Machine and immediately save your renumbered program on disk.

OPTION #5

RESTORING APPENDED CODE FROM DISK

If you have exercised option #3 to remove the machine code from your program, or if you are just starting a new program, to which you wish to add a pre-assembled mini-library, you may use Option #5 (RESTORE APPENDED CODE FROM DISK) by following the procedures listed here:

1) If Routine Machine is in memory, re-enter with a CALL 2051. If Routine Machine is not in memory, enter BRUN ROUTINE MACHINE.

2) When the menu appears, select Option #5 by pressing '5'. You will be asked to confirm by pressing 'Y' (for Yes) or RETURN alone.

3) The program will then display:

```
IS THE MACHINE CODE TO BE RESTORED
CONTAINED IN DISK FILE 'A.M.C.'?  [Y]
```

If you have not used this file name to save the appended modules, enter 'N' (for NO). It will then display:

```
WHICH FILE NAME THEN? (<RET> = QUIT)
(FOR CATALOG ENTER 'CAT')
```

You may now either catalog the disk or enter the name under which the modules were saved.

4) Routine Machine will now restore the machine code which was formerly copied to disk, and return you to the menu.

If the appended machine code was originally copied to a disk file under the name 'A.M.C.', then Routine Machine will delete this file from the disk. If you specified some other name for the disk file, then it will not be deleted.

When restoring appended code from disk (unlike the case of appending a new routine) it is NOT necessary that the disk be an S.D.S. disk.

NOTE: A possible conflict can arise between the use of the ampersand to invoke Routine Machine modules and its use to invoke utilities such as Apple Computer's Renumber routine. In the previous section it was explained how to use Renumber on an Applesoft program with appended modules.

The point to note there is that the Renumber routine could be used normally because the Applesoft program had not been run, and so the ampersand vector had not been reset. Sometimes, however, you may wish to have an ampersand-invoked utility available both before and after running your own program. The problem is that when you run your program the ampersand vector will be set to the Interface Routine at the end of your program, and will then no longer be pointing to your utility. What to do?

There are two solutions to this problem. The first follows from the fact (surprise!) that it is not necessary to use the ampersand to invoke Routine Machine modules. It is possible to invoke modules using a CALL directly to the address where the Interface Routine is located. This method is discussed in detail in Appendix B, "Alternative Methods of Invoking a Module". If this method is used then the ampersand vector is not reset when your program is run (unless your program resets it in some other way), and so your ampersand-invoked utility is always connected.

But normally the method of invoking modules by means of the ampersand is preferable. In this case we may employ a Routine Machine module to solve the problem. When your program is run, the line which causes the ampersand vector to be reset to the Interface Routine also causes the old ampersand vector to be saved. If you have the routine RESTORE AMPERSAND.RM appended to your program, then it should be invoked just before your program ends, and it then has the effect of restoring the original ampersand vector. Thus on return to immediate command mode your ampersand-invoked utility will have been reconnected and will be ready to be used.

If the RESTORE AMPERSAND.RM routine is used, it MUST be the LAST module invoked before your program comes to an end. If not then a subsequent module invocation will have the effect of invoking your utility from within your program, with unpredictable consequences.

OPTION #6

REPORT MODULES APPENDED

Routine Machine may be used to append up to 255 modules to an Applesoft program. Although it is unlikely you will even approach this number, you may from time to time wish to see a list of the routines which have been appended to a particular Applesoft program.

Pressing ´6´ at the main menu will produce a Module Report (provided that there are any modules to report on). This is useful because it allows you to see what modules have already been appended and to check on the names given to them. Sometimes you will wish to re-append a module, in which case you can determine the name of the disk file from which it was originally taken.

When selecting each of the ´report´ options 6 - 9, you are asked if output is to go to the printer. If you simply want screen display, press RETURN (or ´N´). If you want a print-out, press ´Y´. The Routine Machine will then ask:

PRINTER IN SLOT: 1

If this is the correct slot for your printer, press RETURN, otherwise enter the slot your printer is in (the Routine Machine will remember).

For each module appended you are informed of the invocation name and the name of the disk file from which the subroutine was originally obtained. No information is provided as to the addresses of the modules in memory, since you need not be concerned with this. In any case, these addresses change each time a line or statement is added to or deleted from your Applesoft program.

For example, using Option #6 of the Routine Machine might give the following information:

```
          *** MODULE REPORT ***          [3]

      INVOKED AS:       FILE NAME:

#1    TONE              BEEP.RM

#2    INPUT             STRING INPUT.RM

#3    ERR               ERR.RM

      PRESS 'S' TO SEARCH PROGRAM
         OR <RETURN> FOR MENU
```

The Module Report displays eight modules at a time. If you have more than eight modules appended then press the space bar to go to each successive page (or you can return to the menu at any point by pressing RETURN).

It is also possible to transfer directly from the Module Report to the Search facility by entering 'S'. (See the next section for detailed information on the Search option.)

OPTION #7

SEARCHING FOR CALLS AND AMPERSANDS

Normally modules will be invoked using the ampersand, but there is an alternative method which employs a CALL directly to the address of the Interface Routine (see Appendix B for the details). Thus when searching for the invocation statements in your program you may specify that the search is to be for ampersands or for CALLs. (The latter sub-option also allows you to search for all CALLs in an Applesoft program whether or not they are module invocations.) You can toggle between these two sub-options by pressing 'T'.

Having decided whether you are interested in CALLs or in ampersands, you must now specify whether you are interested in (i) all statements in your progam employing an ampersand (or CALL), (ii) all module invocations, or (iii) only invocations of a particular module.

To see how the SEARCH option works, follow these examples:

1) From Routine Machine, select Option #0 to exit the program. Then type in 'NEW' to clear memory.

2) Now enter LOAD BINARY FILE COPIER (on the backside of the Routine Machine disk).

3) When the Applesoft prompt returns, enter CALL 2051 to re-enter the Routine Machine.

4) At the menu, select Option #7 and press RETURN. Routine Machine will then display the following:

```
SEARCH TYPE:   AMPERSAND STATEMENTS

DO YOU WISH TO SEARCH FOR:

1  ALL SUCH STATEMENTS?
2  ALL MODULE INVOCATIONS?
3  INVOCATIONS OF A PARTICULAR MODULE?

(PRESS 'T' TO CHANGE SEARCH TYPE,
OR <RETURN> TO QUIT)
```

5) Select Option #1 to display all of the ampersand statements. Since BINARY FILE COPIER contains a considerable number of ampersand statements it will be necessary to press the space bar once or twice to display them all (remember only eight are displayed at a time). Notice that on each page the Routine Machine displays:

```
PRESS 'S' FOR NEW SEARCH
   OR 'L' TO INSPECT LINE
   OR <SPACE> TO CONTINUE
```

at the bottom of the screen to prompt you for the next page of ampersand statements, as well as to provide you the option of inspecting a particular line or to change the search to CALLS instead of AMPERSANDS.

Since the ampersand is used in the program only to invoke modules, and this is the only method used to invoke modules, you will see the same display whether you select option 1 or option 2.

6) Now press 'S' to return to the search menu, and press 'T' to change to the search for CALLS and repeat the steps to display the CALLS used in the program. Notice the top line of the search menu now appears as:

```
SEARCH TYPE:   CALL STATEMENTS
```

When an ampersand or CALL statement is found which satisfies the conditions of the search, it is displayed along with the number of the line in which it occurs. If the line consists of more than one statement, as in the statement below:

```
N = INT(VAL(LEFT$(B$(J),4))): FOR I = 1 TO N:
&"TEXT", A$(I);: PRINT " = "A(X(I)): PRINT: NEXT
```

then only the invocation statement itself will be displayed, as in:

```
& "TEXT", A$(I);
```

SPECIAL NOTES:

If you are using the CALL method of invoking modules (see Appendix B), a typical invoking statement would be:

```
CALL IR "NAME", A$, B$
```

You might erroneously omit the ´IR´ so that your program would contain the statement:

```
CALL "NAME", A$, B$
```

This would make a nice method of invoking a module, except that "NAME" (a string literal) cannot be evaluated as a calling address.

If you use the SEARCH OPTION, and Routine Machine finds a CALL statement with some expression following the CALL which cannot be evaluated as an address, it will display the offending statement along with an error message, and return you to immediate command mode. This gives you the opportunity to correct the CALL (by replacing, e.g., CALL "NAME" with CALL IR "NAME"). You can then re-enter Routine Machine with the usual CALL 2051 and resume normal operations.

The Routine Machine may be brought to bear on ANY Applesoft program, not only to those with Routine Machine modules. Thus you can use the Routine Machine on any program to answer the following questions:

a) How long is it?

b) Does it have appended machine code?

c) Where does it end?

d) Does it employ ampersand statements, and if so, in which lines?

e) Does it employ CALL statements, and if so, in which lines?

f) Does it have unusual bytes embedded in lines (especially in REM statements)?

OPTION #8

INSPECTING APPLESOFT PROGRAM LINES

This option is by no means required for the normal use of Routine Machine, but may prove useful to those wishing to write their own ampersand routines. It's main utility is in being able to see the exact contents of a line, as it is actually stored in memory. See the "Concluding Notes on Writing Your Own" for more details on the use of this option when designing your own ampersand routines.

The INSPECT LINE option is entered from the main menu by pressing '8'. It can also be reached directly from the SEARCH option by pressing 'L' at the appropriate point. You may specify a line number and (if the program contains a line with that number) the line will be displayed together with the actual bytes in memory which constitute that line, along with its ASCII representation. You can move from line to line by entering 'N' for Next Line. If you are unsure of the first line number, enter '0' as the number to look for, then 'N' for the next line.

Occasionally, if a line is very long, the display will be too large to fit on the screen all at one time. In this instance Routine Machine will display as much as it can, then beep discreetly and wait for you to press a key to display an additional line. It may take a few keystrokes to bring up the whole display.

SPECIAL NOTES:

When you exercise the INSPECT PROGRAM LINE option, the bytes which constitute the line will be displayed, together with a partial representation of them in ASCII form. The bytes making up the link field and the line number are shown separately from the bytes making up the text of the line itself. (Those unfamiliar with the structure of an Applesoft line may turn to Appendix B of this manual for an explanation.)

In the hex dump of the text of the line, some bytes are shown in inverse. These are the Applesoft tokens, and all have values in the range from $80 to $EA (for a complete list, see p. 94). By comparing the hex dump to the Applesoft line displayed at the top of the screen you can see how the line is tokenized.

In the ASCII representation to the right of the hex dump on the screen you will observe that some bytes are represented by normal characters, some by inverse characters and some not at all. The inverse characters are representations of text in the program line (text always consists of bytes with values in the range $20 to $5A). Most Applesoft tokens represent reserved words, such as PRINT ($BA) and CALL ($8C), and such token bytes are not represented in the ASCII dump. When a token such as $CF (´<´) can be represented with one byte, its ASCII equivalent is shown in normal text.

In the case of the statement:

```
10  A$ = "="
```

the first "=" is tokenized as $D0, whereas the second "=" is represented in the line by $3D. In the ASCII representation of this line, the first "=" will appear normally and the second "=" in inverse.

The INSPECT PROGRAM LINE option allows you to locate an Applesoft line in memory and then exit directly to Monitor to make changes in the line.

For example, you could put the following line in a program:

```
10  PRINT "APPLE ]Z PLUS"
```

then locate the line using Option 8 and change it to:

```
10  PRINT "APPLE ][ PLUS"
```

To see how this is done, try the following:

1) Type in ´FP´ to clear memory, then enter line 10 as first shown above as a simple program.

2) BRUN ROUTINE MACHINE and select Option 8 to view the line in memory. The following display should be shown:

```
10 PRINT "APPLE ]Z PLUS"

LINK FIELD AND LINE NUMBER:
2601- 16 26 0A 00

BYTES COMPRISING TEXT OF LINE: ´´´´´´´´
2605:                  BA 22 41         "A
2608: 50 50 4C 45 20 5D 5A 20  PPLE ]Z
2610: 50 4C 55 53 22           PLUS"

END-OF-LINE TOKEN (00) AT $2615

END OF PROGRAM

PRESS ´L´ TO INSPECT ANOTHER LINE
   OR ´*´ TO EXIT TO MONITOR
   OR <RETURN> FOR MENU
```

3) Note location $260E holds the ´$5A´ that corresponds to the ´Z´ in the statement.

To change the $5A (´Z´) to the $5B appropriate to the ´[´ character, first enter the Monitor by pressing the asterisk (´*´), then type:

```
260E: 5B
```

and press RETURN. From the Monitor you can now re-enter Routine Machine by typing: 80BG and pressing RETURN.

List line 10 again and you should now find that the line lists as:

```
10  PRINT "APPLE ][ PLUS"
```

This exit-to-Monitor feature allows you to introduce "illegal" line numbers into your program. Due to the peculiarities of Applesoft (and specifically the LINGET routine at $DA0C), Applesoft will not accept line numbers greater than 63999. (This is due to the algorithm used by LINGET to convert a decimal number to a hexadecimal number.) To give a line the number ´65535´, for example, proceed as follows:

1) Locate line 10 as previously entered. Note that the line number bytes (0A 00) are held in locations $2603,2604. Enter the Monitor as before, and type:

2603: FF FF

and press RETURN. Now re-enter Routine Machine and display the line 65535. You'll notice that such a line is now present, and it is in fact, our old line #10.

Such a line cannot be deleted from an Applesoft program by normal means, and so is useful for placing copyright statements, etc. If you should later decide to edit the line, you can use Routine Machine to locate the line, and change the FF FF byte-pair to FF F9. This will then list as line number 63999, which can be edited as you desire, directly from the keyboard.

When changing the line number field, you must be careful not to change the link field accidentally. If you do, then the program will LIST as garbage from this line on. In such a case, you could use Routine Machine to repair the damage. Alternatively you could simply enter: DEL 1,0. This rather unusual statement will not delete any lines, but will force Applesoft to tidy up the link fields for every line.

Another interesting experiment is to change the link field of a line to point to the line AFTER the next. For example, consider the program:

```
10 PRINT "HELLO"
20 PRINT "WORKING..."
30 PRINT "BYE."
```

The link field of line #10 normally points to the link field of line #20. This could be modified to point instead to the link field of line #30. Line #20 would then disappear from the listing, but would still execute in a running program! Unfortunately, any attempt to add or delete lines, or to re-LOAD the program after a SAVE will restore the hidden line. Oh Well ...

OPTION #9

THE MEMORY MAP

The use of this option is not required for appending or removing modules, but does provide useful information such as the number of bytes of machine code appended to an Applesoft program.

The Memory Map is selected by choosing Option #9 from the main menu of Routine Machine. As with previous 'report' options, you may output the Memory Map to the printer by entering a 'Y' when Routine Machine asks you about printer output.

As noted before, the Routine Machine occupies memory from $800 to $25FF, and when it is present your Applesoft program starts at $2601 instead of the usual location of $801 (see the diagrams on pp.13-15 of this manual). Normally you will be more interested in the location of your program at run time, rather than its location when the Routine Machine is present. Thus when the Memory Map is first displayed it shows the location of your program AS IF it were at $801. If you then press 'A' the Memory Map will change to display the current actual addresses (with your program at $2601). You can toggle back and forth between these two modes using the 'A' key.

The Memory Map does not always have the same format. It differs according to whether there is a program in memory and whether it has any appended modules.

When there is no program in memory, the Memory Map displays only four addresses, such as the following:

```
PROGRAM START:          $0801 =  2049

LOMEM:                  $0804 =  2052

HIMEM:                  $9600 = 38400

DOS BUFFERS:            $9600 = 38400
```

Considerably more information is displayed when there is a program in memory. The following is a typical Memory Map, corresponding to a situation like that shown in Figure 3 on page 15 of this manual. (It was in fact done from the SORT DEMO 1 program on the Routine Machine diskette.)

```
                APPLE ][ MEMORY MAP

PROGRAM START:                  $0801 =  2049
BASIC PROGRAM END:              $0F5F =  3935
ACTUAL PROGRAM END:             $1289 =  4745
LOMEM = SIMPLE VARIABLES:       $1289 =  4745
ARRAY SPACE:                    $12BA =  4794
ARRAY SPACE ENDS:               $12E2 =  4834

STRINGS:                        $936E = 37742
HIMEM = END STRINGS:            $9600 = 38400
DOS BUFFERS:                    $9600 = 38400

BASIC PROGRAM LENGTH:           $075E =  1886
BYTES APPENDED:                 $032A =   810
TOTAL PROGRAM LENGTH:           $0A88 =  2696
FREE SPACE:                     $808C = 32908
STRING STORAGE:                 $0292 =   458
NOTHING UNDER DOS BUFFERS

MAXFILES = 3

R = MODULE REPORT
A = ADDRESS TOGGLE, M = MENU, B = BASIC
```

The program start is the location in memory where your Applesoft program begins or, if there is no program in memory, where an Applesoft program will be placed if one is LOADed. This is normally $801 (decimal 2049), but (as stated above) if Routine Machine has been installed, it will be $2601 (decimal 9729).

When you BRUN ROUTINE MACHINE, any program which is at the normal location of $801 is relocated to $2601. Your program will function normally at this address however, and you may SAVE and LOAD Applesoft programs in the usual way. (NOTE: The only exceptions are programs which use the Hi-Res pages. See the opening section, "A Little More Detail" for more information on this.)

Since HIMEM on a 48K system is normally set to $9600, this leaves 28K of RAM for your Applesoft program to RUN in. By setting MAXFILES to 1, the available RAM can be extended to almost 30K. On exiting Routine Machine, you can always recover the 7.5K that it occupied by BRUNing REMOVE ROUTINE MACHINE. This utility will relocate your Applesoft program from $2601 back to $801.

Most users of Routine Machine will not be working with programs larger than 28K. If, however, you have a large program and a number of other large utilities such as Apple-Doc, A.C.E., etc. installed in the upper reaches of memory, there arises the possibility of exceeding HIMEM when appending a lengthy subroutine. Since this is obviously undesirable, some care should be taken by the user to make sure that there is enough memory for the module to be appended. In the event that you do try to append a module which would result in the length of the program exceeding HIMEM, a PROGRAM TOO LARGE error will be displayed, and you will be returned to the main menu. It is also possible for this to occur when restoring modules from disk (Option #5).

Although it is most unlikely that this will ever become a problem, if you have doubts, a periodic check of the memory map will display the remaining free memory space.

The program start is determined by looking at the start-of-program pointer, TXTTAB, at $67,68 (decimal 103,104), and is also the address of the first link field of the first line of your Applesoft program. (This must be preceded by a $00, so if you reset TXTTAB to some other address, be sure to put a $00 at TXTTAB-1.)

The actual program end is determined by the contents of $AF,B0 (decimal 175,176) which is the end-of-program pointer (PRGEND). This pointer indicates the address of the byte immediately following the last byte of your program (including any appended machine code).

The BASIC program end is the address of the byte immediately following the double-$00 which follows the last line of your Applesoft program. If you have Routine Machine modules appended, then this is the address of the first module.

The values for the BASIC program length, the number of bytes appended and the total program length are all obtained from these three addresses.

LOMEM is the address of the lowest address available for use by the Applesoft program currently in RAM. Normally it is the same as PRGEND. LOMEM is the address at which are stored the names and values of the simple variables used in your program. Following the simple variable space is the space used to store the names and values of the array variables used in your program.

For more details on these matters you should consult the article "Ultimate Input-Anything Routine" in the first issue of Call-A.P.P.L.E. In Depth (Sept., 1981), pp 94-99. (NOTE: This issue also contains a number of other extremely valuable articles, including a reprint of the article by John Crossley on Applesoft internal entry points, which are helpful in creating your own ampersand routines. This book is available through S.D.S.)

When an Applesoft program inputs character strings, or when it constructs strings, it stores them from HIMEM down. The pointer FRETOP (at $6F,70) points to the beginning of this string storage area. Strings are stored from FRETOP up to HIMEM, and the difference between these two addresses is given in the memory map as 'string storage'. The difference between the start of string storage, FRETOP, and the end of the array space, STREND ($6D,6E) is given in the memory map as 'free space'. Note that this value changes when you toggle between the two modes of the Memory Map.

Option #9 thus allows you to determine the amount of memory used by an Applesoft program for its simple variables and for its arrays. (To enter Routine Machine, directly from an Applesoft program without clearing variables, enter the Routine Machine with a CALL 2055, rather than a CALL 2051. This will skip the main menu and preserve current variable pointers.)

It also allows you to determine the quantity of strings generated by a program. As new arrays are dimensioned, STREND is raised, and as new strings are input (or generated), FRETOP is lowered. When these meet (or when they almost meet), the amount of free space is reduced to near zero. At this point during the execution of a program, Applesoft takes time off to get rid of any strings in the storage area no longer required (such as intermediate strings generated in the course of string manipulation).

This 'garbage collection' occurs periodically during the running of any Applesoft program that generates many strings. When it occurs, the FRETOP pointer is raised and then is progressively lowered as further strings are generated. This must be remembered if the Memory Map is to be used as a guide to the quantity of strings generated by an Applesoft program.

Note: This process can greatly slow down Applesoft programs with large string arrays. Ampersoft Program Library, Vol. 1 contains routines to speed this process up by as much as 40 times.

Even with a memory map however, there may be some uncertainty concerning the quantity of strings generated by an Applesoft program, normally no such uncertainty surrounds the amount of memory taken up by the numeric variables. Once a simple variable is defined, or an array is dimensioned, the memory allocated cannot normally be de-allocated, unless you are using the array delete routine from Ampersoft Program Library Volume 1 (array related routines).

On a 48K system, DOS itself (as distinct from its buffers) occupies the range from $9D00 to $BFFF. Immediately below the DOS routines (or more accurately, immediately below, except for seven unused bytes at $9CF9-9CFF) one normally finds the DOS file buffers. These are buffers used by DOS during disk I/O. (See pages 6-12,6-13 of the book Beneath Apple DOS from Quality Software for a full description of the DOS file buffers.)

The address of the first DOS buffer is stored at $9D00,9D01, and the number of DOS buffers is stored at $AA57. The DOS buffers may be lowered by placing a new address at $9D00,9D01 and then calling the DOS buffer construction routine at $A7D4. This is the method used to install a utility program such as A.C.E. (Applesoft Command Editor from Southwestern Data Systems) or Program Line Editor (Synergistic Software) between DOS and its file buffers.

The address given as ´DOS FILE BUFFERS´ in the memory map is the address of the bottom of the DOS file buffers. This is normally $9600, but if A.C.E. or P.L.E. is installed then it can be lower. It is of course, altered when the value of MAXFILES (the number of DOS buffers) is changed.

You can determine the address of the DOS file buffers for different values of MAXFILES quite easily. With the Routine Machine installed (and a program in memory), CALL 2055 to get a memory map. Exit by pressing ´B´ for BASIC and change the number of DOS buffers with a command such as MAXFILES 1 (the minimum) or MAXFILES 16 (the maximum). Then CALL 2055 to get a new memory map, and you will see how the address of the DOS file buffers has changed. Since the Routine Machine resides below your Applesoft program, the value of MAXFILES (and also the value of HIMEM) can be changed without interfering with the operation of the Routine Machine.

Normally HIMEM is the same as the address of the bottom of the DOS file buffers. Many utilities are designed to reside in the higher reaches of memory, just BELOW the DOS file buffers (or just below any other utilities which are already there). When such a utility is installed, HIMEM is normally lowered to protect it from being clobbered by a deluge of strings when the Applesoft program is run. When HIMEM is reset in this way, the memory map will show the number of bytes between the DOS buffers and HIMEM.

If you have Apple Computer's Renumber program then the following may prove instructive:

First enter an 'FP' to set HIMEM to its usual $9600 (assuming MAXFILES = 3). LOAD Renumber, but do not RUN it. BRUN ROUTINE MACHINE and you will see that the Renumber has 2306 bytes of machine code appended. (At this point you might search the program for CALL statements.)

Now exit the Routine Machine and RUN the program. CALL 2051, get a memory map, and you will see that there is no longer any program in memory, but that HIMEM has been lowered to $8E00 and there are 2048 bytes below the DOS buffers. This is the machine language routine which renumbers your program. The 258 bytes of machine code which have disappeared were required only to relocate the main routine from the end of the BASIC program to the upper reaches of memory.

APPENDICES

APPENDICES

>> APPENDIX A <<

MAKING BACK-UP COPIES

The Routine Machine diskette cannot be copied using ordinary copy programs. It does, however, come with its own copy program with which back-up copies of the diskette may be made.

It is HIGHLY RECOMMENDED that you DO NOT USE YOUR ORIGINAL diskette for daily use. Upon purchase, make a backup copy immediately and put the original in a safe place.

To run the copy program, place the Routine Machine diskette in your disk drive and type in: PR#6 to boot in the usual manner. When the red drive light comes on, immediately press the ´C´ (for ´Copy´) key on your Apple. The program will then confirm that you wish to make a copy, at which point you should enter the values for the SINGLE DRIVE which you want to use to make the copy. The copy process is restricted to a SINGLE DRIVE COPY to assure maximum reliability in the copy. You will, however, have to exercise due caution in exchanging the diskettes during the operation in order to make sure the appropriate diskette is in the drive at that moment. If you think you have made a mistake at any point, stop IMMEDIATELY and re-boot to restart the process.

The copy program will produce a total of three back-up copies of the Routine Machine diskette. This gives you a total of four working copies of the program. In addition, if you should ever accidentally erase a particular file from a given diskette, you may use the ´FID´ file transfer utility provided on your Apple System Master diskette to move the particular file from your original Routine Machine diskette to the diskette from which the file was erased. This in no way affects the counter on the master back-up copy program.

It is not necessary to recopy the entire diskette should only a given file become damaged.

NOTE: If you have an Integer Apple with Language or RAM card-loaded Applesoft, first boot on the Apple System Master, as originally supplied with your disk drive. Then re-boot with a PR#6 on the Routine Machine diskette, following the instructions given above.

>> APPENDIX B <<

ALTERNATIVE METHODS OF INVOKING A MODULE

Please note that this section is not required to make use of the Routine Machine. This section explains a method of invoking Routine Machine modules which does not make use of the ampersand. However, if you have not previously studied the subject, the following discussion of the methods available for appending machine code to the end of an Applesoft program may prove to be of some interest.

When an Applesoft program is present in memory there are two ´pointers´ which point to the beginning and to the end of the program. These are stored on Page Zero ($0000-$00FF) in the usual 6502 convention (low byte followed by high byte) as follows:

OFF

POINTER	HEX ADDRESS	DECIMAL ADDRESS	DESCRIPTION
TXTTAB	$67,68	103,104	Start of Program
PRGEND	$AF,B0	175,176	End of Program

ON

The start of the program could thus be determined from BASIC by the statement:

```
BEG = PEEK(103) + 256 * PEEK(104)
```

and the end by:

```
END = PEEK(175) + 256 * PEEK(176)
```

(NOTE: PTR READ.RM and PTR WRITE.RM are Routine Machine library routines which make such determinations quite simple. See the section on those routines for more details.)

To understand better how Applesoft actually stores a program in memory, type ´FP´ and enter the following program:

```
 5  A = 1: B = 2
10  C = 3
```

Now go into the Monitor with a CALL -151. Type 67.68 and press RETURN. The computer should print out:

```
0067- 01
0068- 08
```

Now type AF.B0 and press RETURN. The computer will again respond, this time printing:

```
00AF- 17
00B0- 08
```

The first pair of numbers (taken in reverse order) means that the program starts at $0801 (decimal 2049), and the second pair means that it ends at $0817 (decimal 2071).

Now enter 800.817 and the bytes which constitute the entire Applesoft program will be displayed as follows:

```
0800-  00 0D 08 05 00 41 D0 31
0808-  3A 42 D0 32 00 15 08 0A
0810-  00 43 D0 33 00 00 00 ??
```

Although the program starts at $801, a $00-byte must occur at $800 (otherwise the Applesoft Interpreter will get upset).

The next two bytes, 0D and 08, point to the location in memory of the next line. These two bytes, or more generally, the first two bytes of any program line, are sometimes referred to as the 'link field', since they link that line to the line which follows (if any).

The next two bytes hold the number of the line, in hexadecimal form, low-byte first, as usual. In this case we have 05 00, that is, $0005 (decimal 5). The line number of the second line is represented at $80F,810 by 0A 00, that is, $000A (decimal 10).

At location $805 begins the storage of the actual text of program line #5. Applesoft conserves space by 'tokenizing' each program word, where possible. That is, words such as 'PRINT' will be stored as a single byte. Spaces between keywords are always eliminated from program lines (although, of course, spaces are not eliminated within PRINT statements, or in a REMark). Examining locations $805 to $80B, you will find the bytes which correspond to 'A = 1: B = 2' (namely: 41 D0 31 3A 42 D0 32). Terminating the line is the end-of-line token, $00.

The second line possesses a similar structure. The program ends with a double-$00 at $815,816. This is the location of what would have been the next link field, pointed to by the first two bytes of the second line of our sample program. This in fact provides one way of identifying the end of an Applesoft program, since you can scan the hex data for a triple zero. (The first zero of the triple zero, you'll remember, is the end-of-line token of the last line of the program.)

When the Applesoft interpreter finds a double zero where it is looking for a link field, it knows that it has reached the end of the program. In the example above, the final end-of-line token is at $814, and the double zero at $815,816. Note that PRGEND ($AF,B0) points to $817, the byte immediately after the last byte of the triple zero (here shown by '??', since it could be anything). Generally pointers to the end of memory blocks point, not to the last byte of the block, but to the byte immediately following.

During the execution of a program, the Applesoft interpreter does not check PRGEND to locate the end of the program. Instead, it simply stops when it comes to the double-zero link field. On the other hand, the SAVE command in DOS does not care where the double-zero (which marks the end of the BASIC portion of the program) is, but simply checks PRGEND to get the address of the end of the program.

This means that we can include extra bytes of information following the double-zero, but before an artificially-high setting of PRGEND, and this information will be SAVEd along with the program. This is how it is possible to append a machine language subroutine to an Applesoft program so that it becomes a permanent part of the program, which can then be SAVEd and LOADed along with the BASIC portion of the program.

Once a routine has been appended in this way, the question arises as to how it is to be called or invoked by the Applesoft program (or more accurately, by a statement within the Applesoft program). If the routine is to be appended to a finished Applesoft program, and that program is always run at the same address in memory, then you could conceivably CALL the routine at a fixed address. This method, though, is not very practical, since programs are often changed, with the result that the absolute memory locations of any appended routines change accordingly.

Fortunately, there is an alternative. If one machine language routine of length N bytes is appended to the end of an Applesoft program, then the address of the first byte of the routine will always be found at the address of the end of the program minus N. (The term 'program' here means BOTH the BASIC portion together with the appended machine code.) That is to say, one would look at the value contained in PRGEND (175,176) to determine the end of the program 'envelope', and then subtract N bytes.

Thus the routine could be called at the address:

```
PEEK(175) + 256 * PEEK(176) - N
```

This address will remain valid despite any changes in the length of the BASIC portion of the program, since the pointer at locations 175,176 (PRGEND) always points to the end of the program envelope.

This method can be generalized to accommodate more than one appended routine, but to append several routines by hand can be rather tedious. Fortunately, you don't have to, since the Routine Machine will now do it all for you!

Whenever you have added routines to your program using the Routine Machine, there will also be present a machine language routine which handles the initial part of your module invocations. This routine is automatically included in your program when you append the first module.

This routine is known as the 'Interface Routine' because it acts as an interface between your Applesoft program and its appended machine language routines. It is always placed at the very end of the program envelope, and occupies the last 203 bytes. Thus it can be called at the address:

```
IR = PEEK(175) + 256 * PEEK(176) - 203
```

This is the address set up in line #1 of your program by the EXEC file on the Routine Machine diskette named CALL SETUP.

When, during the course of the execution of your Applesoft program a Routine Machine module is invoked (by one means or another), the first thing that happens is that control passes to the Interface Routine, which then looks at the module name (between the quotes in the module invocation) in an attempt to find out which particular module you're interested in. With the name in hand it then searches through the modules appended, and if it finds the one you want then the Interface Routine passes control on to that routine.

USING THE AMPERSAND TO INVOKE MODULES

Although the Interface Routine can be called directly in a statement such as:

```
CALL IR "INPUT", A$
```

there is a more elegant method, using the ampersand ('&').

At locations $3F5-3F7 (decimal 1013-1015) is a JMP instruction (see the Apple][Reference Manual, p. 132). If the address of the Interface Routine is placed into locations 1014 (low byte) and 1015 (high byte), then when the Applesoft interpreter encounters the ampersand control will pass to the Interface Routine. As before, the routine will then locate the routine that you wish to invoke, and pass control to it.

Thus in this system there are TWO distinct ways of invoking appended routines. The effects are the same, and only the syntax of the module invocations varies. Below are examples of module invocations using each of the two methods:

CALL(768)X,Y

```
CALL IR "SWAP", A$, B$
     & "SWAP", A$, B$

CALL IR "SORT", A$(FE) TO A$(LE)
     & "SORT", A$(FE) TO A$(LE)
```

The two methods have the same result: To invoke the Interface Routine, which then reads the name of the module, locates it in memory (somewhere between itself and the end of the BASIC portion of your program), and then JMPs to the routine.

If the CALL method is used, it is necessary to have a statement in your Applesoft program which defines the CALL address 'IR'. Such a statement can be inserted in your program (as line #1) by EXECing the file CALL SETUP on the Routine Machine diskette (with your program in memory, of course).

If the ampersand method is used, then the ampersand vector must be set up to point to the address of the Interface Routine. This can be done by a CALL to an internal entry point in the Interface Routine itself. This internal entry point is 46 bytes before the end of the Interface Routine, which is itself at the end of your Applesoft program, and so the required ampersand vector setup can be accomplished with a simple

```
CALL PEEK(175) + 256 * PEEK(176) - 46
```

The file AMPERSAND SETUP on the Routine Machine diskette can be EXECed to insert this line (as line #1) in your program. Then when your program is run the ampersand hook-up is automatically made and your program can then invoke appended modules without further thought.

It is up to you which method you wish to use to invoke modules. The ampersand method is more pleasing to the eye, and also involves fewer keystrokes, but it does modify the ampersand vector which may be used by other utilities. If you wish to use ampersand utilities, you may care to use the CALL IR method of invoking modules. It is also possible that you might have an ampersand routine that is entirely independent of the Routine Machine. In that case, the CALL IR method will leave the ampersand free for other duties.

If you are still unsure as to the difference between these two methods of invoking modules, then simply ignore the existence of the CALL method, and stick to ampersands (they'll never let you down).

>> APPENDIX C <<

GETTING SLIGHTLY MORE TECHNICAL ...

The routines which can be appended to an Applesoft program using the Routine Machine are NOT limited to those available on the Routine Machine and Ampersoft Program Library disks. You can also use routines which you may find in any of the numerous Apple-related computer magazines or routines which you create yourself (or have created for you).

PUBLISHED MACHINE LANGUAGE ROUTINES

If you should decide to use a routine listed in a magazine, chances are good that it will work with little or no modification with Routine Machine.

The main things to keep in mind when selecting additional routines are:

1) Is the routine relocatable?

Machine language programs can be written in two basic forms in regards to this topic. Any program which contains references to absolute memory addresses within itself is not relocatable. This can usually be seen by JSR's and JMP's to INTERNAL locations in the listing. Also, the text of the article accompanying the routine may say something like "this routine must be run at location $300 to function properly". If the text indicates that the routine may be placed ANYWHERE in memory, the program is relocatable, and is a good candidate for inclusion in your Routine Machine library.

2) Can the routine be called from a running Applesoft program?

Routines should be executable from within a running Applesoft program without any chance of conflict with Applesoft. The main danger here is that you might come across a routine designed to be called from an Integer BASIC program. In such a case is is possible that certain memory locations used by the routine would be required for use by Applesoft.

3) Is the routine invoked with the ampersand?

If so, it will usually work fine as a Routine Machine module (again, provided that it is relocatable). But beware of routines that require POKEs or other memory setups prior to invoking them.

If the routine satisfies these three criteria then you should have no difficulty in using it as a Routine Machine module. If changes are required, or you're inclined to tinker a little, see the following sections for some useful hints.

Note that most ampersand routines do not expect a comma as the first character in the parameter list, whereas a comma IS (usually) the first character in a Routine Machine module invocation (unless, of course, the module invocation has NO parameter list). However, because of a subtle feature of the Interface Routine, the leading comma in a Routine Machine module invocation is generally optional.

What this means regarding the adaptation of ampersand routines taken from magazines is as follows: Suppose, for example, that you find in a magazine a routine to swap string variables, and that the routine is invoked by the command & A$, B$. Then the same routine can probably be used as a Routine Machine module and can be invoked by EITHER of the following commands: & "SWAP" A$, B$

& "SWAP", A$, B$

This is because the Interface Routine gobbles any comma occurring immediately after the module name. (For further details on this point see the section "Concluding Notes on Writing Your Own", p.92 of this manual.)

If a published routine expects to find a comma as the first character of the list, then it will normally use a JSR CHKCOM (hex: 20 BE DE) at the start. This should be omitted if the routine is to be used as a Routine Machine module, because the comma will be gobbled by the Interface Routine.

IMPORTANT: If you intend to use published routines in your own programs for your own personal use, magazines are a fine source of material. If you add the routines to programs which you intend to sell commercially, it is highly recommended that you contact the program authors to arrange their consent and cooperation. Many listings from magazines include the phrase "Commercial Rights Reserved" or similar. This must be taken into consideration when including the routines in your own commercially distributed software.

>> JUMP FILE CREATE <<

There is a wealth of relocatable routines available for performing all kinds of tasks (for example, there are thirty routines on the Routine Machine diskette and an abundance of further routines to follow on the Ampersoft Program Library diskettes). Thus normally you will never have to be concerned with interfacing any non-relocatable routine with an Applesoft program. However, it may be that you already have a routine which you have been invoking at a fixed address and which is not relocatable. Such a routine cannot be used directly as a Routine Machine module. There are, however, two possible solutions.

The first is to take your routine and make it relocatable. If this cannot easily be done then there is a utility on the Routine Machine diskette called JUMP FILE CREATE to provide aid and solace. Using this utility you can create an intermediate calling routine which will function as a further interface between the Interface Routine (appended to your Applesoft program) and your non-relocatable routine (occupying its required position in memory).

The only requirement for using JUMP FILE CREATE is that you know the address of the routine which would normally be CALLed (or invoked via an ampersand). For example, let's suppose that the file BEEP.RM were a non-relocatable routine which MUST be loaded at location $300 (decimal 768) to function properly. To create an interface module, run JUMP FILE CREATE.

You will then be presented with the opportunity to enter the call address in decimal. If you do not know the decimal address, press RETURN and a hex option will be presented. For our example, enter '300' as the hex address. It will then print out the decimal equivalent as a matter of information, and instruct you to press 'S' to save the file. It will then be saved to the Routine Machine disk as 'JMP.$0300/768.RM'. This routine could then be appended to your program like any other routine, using the Routine Machine. If when appending the JMP file, you specify "BEEP" as the invocation name, then your program could invoke BEEP.RM as follows:

```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 PRINT CHR$(4);"BLOAD BEEP.RM, A$300"
20 INPUT "PITCH,DURATION?  "; P, D
30 & "BEEP", P, D
```

If you wish to write your own machine language routines, Routine Machine is an ideal way to interface them to an Applesoft program. The following sections provide a variety of information on different aspects of creating your own routines. Although we can by no means present a tutorial on machine language programming, some of the information may prove valuable for making use of routines which already exist within the Applesoft interpreter (as well as those to be found in the Monitor and in DOS).

In addition, the following aids are available through Southwestern Data Systems:

* MERLIN, an extremely powerful, yet easy to use macro assembler. Besides being one of the best assemblers currently available for the Apple, MERLIN offers additional features like Sourceror, which disassembles raw binary code into pseudo source files, and a fully labeled and commented source listing of Applesoft BASIC. When used with a language card and an Apple Plus, MERLIN becomes a co-resident assembler with the ability to easily switch between MERLIN and Applesoft.

* MUNCH-A-BUG, an excellent de-bugging package for machine language programs. Allows use of labels and includes its own miniassembler. Best of all, it can be set in a "waiting" condition to go into the trace mode only after a given routine has been called, thus allowing the user to trace machine code from within fully operational Applesoft programs.

* ASSEMBLY LINES: THE BOOK, is a beginning tutorial on machine language programming. There is a section on creating relocatable code, and also other topics such as sound generation and disk I/O. See also the continuing series ASSEMBLY LINES in Softalk Magazine. In particular, the early issues of 1982 present specific information on interfacing Applesoft to machine language, and techniques for passing parameters.

* APPLESOFT IN DEPTH, is an extensive study of Applesoft and related routines. Published by CALL -A.P.P.L.E., this is an invaluable reference book for programmers.

On the other hand, if you are unable or unwilling to delve into what may seem to you the Terra Incognita of assembly language programming, yet find yourself in need of a routine to do a certain task, then please contact Southwestern Data Systems. If you have an Applesoft program which otherwise performs well except that it is slow in places, then what you may need is a Routine Machine module to replace the offending portion of Applesoft code. S.D.S. will then either include the requested routine on a future Ampersoft Program Library diskette or pass your request on to an independent consultant.

```
**********************************************************
*                                                        *
*  IMPORTANT: IF YOU DO WRITE YOUR OWN ROUTINES,         *
*  PLEASE WRITE SOUTHWESTERN DATA SYSTEMS! WE MAY BE     *
*  ABLE TO INCLUDE THE ROUTINE ON FUTURE AMPERSOFT       *
*  PROGRAM LIBRARY DISKS! WE CAN ALSO SUPPLY INFOR-      *
*  MATION AS TO WHAT ROUTINES ARE CURRENTLY UNDER        *
*  DEVELOPMENT, AND WHICH ARE STILL NEEDED.              *
*                                                        *
**********************************************************
```

AN INTRODUCTION TO THE AMPERSAND

Before looking at how some of the specific routines work, let's first consider how the ampersand vector itself works. Consider this first example:

```
10 PRINT "HELLO"
20 CALL 768
30 PRINT "BYE"
40 NORMAL
```

where, in addition to this Applesoft listing, the following code has been placed at location $300 (768 decimal):

```
0300- 20 80 FE   JSR $FE80
0303- 60         RTS
```

Enter the Applesoft program as listed. Enter the machine language routine by entering:

```
CALL -151
*300: 20 80 FE 60
```

Now RUN the Applesoft program and the words "HELLO" (normal) and "BYE" (in inverse) will appear on the screen.

When a program runs, Applesoft interprets the text of each line by examining each character (or token) on the line, one at a time. It keeps track of where it is at with a pair of zero page bytes, $B8 and $B9 (called TXTPTR for "Text Pointer").

But when a CALL, ampersand (&) or USR statement is encountered, Applesoft is temporarily suspended. In our example, the CALL statement causes a jump to location $300 (decimal 768) at which point is our short machine language routine. At the end of the routine, the usual RTS (Return from SubRoutine) returns control to the Applesoft program. In this regard, the CALL statement is very similar in function to a GOSUB in Applesoft, with the exception that instead of calling an Applesoft subroutine, a machine language routine is invoked.

At location $300, the JSR $FE80 is a call to the routine which puts the Apple screen display in the inverse mode. Once the return to BASIC is made, any further text will appear in inverse, until a NORMAL statement is executed.

Now type the sample program as listed here, and run it.

```
 1 POKE 1014,0: POKE 1015,3
10 PRINT "HELLO"
20 &
30 PRINT "BYE"
40 NORMAL : END
```

Although this program should behave in the same way as the first, the manner in which the call to the machine language subroutine at $300 was done has changed. In this program we have used an alternate feature of Applesoft, the ampersand. The ampersand is different from a CALL in that instead of jumping to an address which would otherwise follow the ampersand, the address for the jump is determined by looking at locations 1014,1015 ($3F6,3F7), called the "ampersand vector".

The disadvantage of this is that the ampersand will always jump to the same location, unless locations 1014,1015 are specifically changed. The advantage is that the ampersand is much more compact than the CALL 768 statement.

In terms of actual program execution, Applesoft's TXTPTR goes directly to line 30 (or the next statement if there was one) after returning from the ampersand call. The question then arises, "How are routine names and parameter passing handled?".

The answer to this question is found in the direct management of TXTPTR by the routine that is called. Consider the statement:

```
20 & "NAME"
```

When Applesoft encounters the ampersand, and passes control to whatever routine is at the address specified by the ampersand vector, TXTPTR is pointing to the first quote of the character string: "NAME". If the called routine does not advance TXTPTR itself and exit with TXTPTR on the ´00´ at the end of line 20 (or the colon if another statement followed), then a SYNTAX ERROR would occur upon return when TXTPTR tried to analyze "NAME" via the continuing Applesoft program. Try modifying line 20 of the sample program as shown above to verify this.

In the case of Routine Machine, the ampersand vector is pointed at the Interface Routine appended to your program. This Interface Routine when called REQUIRES a name to follow the ampersand. By incrementing TXTPTR itself, it can read in the name following the ampersand, and attempt to find the module named among the appended modules. Once the module is found, a jump to the beginning of the routine is done, at which point it is the responsibility of the routine itself to handle any further text following the ampersand and preceding the next colon or end-of-line $00.

For example, consider our simple routine presented earlier. When used directly as an ampersand routine in the first sample listing, the syntax was simple:

```
20  &
```

When used as a routine appended by Routine Machine, the new syntax would become:

```
20  & "NAME"
```

where "NAME" is the invocation name given when the routine is appended to your program. Remember that the Interface Routine will handle the "NAME" before passing control to your routine.

Try saving the code at $300 as a Routine Machine module by placing the Routine Machine diskette in the drive and typing in:

```
BSAVE INVERSE.RM, A$300, L$4
```

This will save the routine which can now be appended to the Applesoft program using the Routine Machine. The routine can then be invoked with the module invocation:

```
20  & "INV"
```

Instead of ´INV´ you could use any other name (though it must be the invocation name you specify when you append the routine).

PASSING PARAMETERS

The next question is: What about passing variables back and forth? As it happens, this is easier than you might at first suppose. With an elementary knowledge of machine language it is possible to take advantage of the routines already present in Applesoft to do most of the work for us. The best way to illustrate this is with a partial reprint of some articles which appeared in Softalk Magazine, in the column ASSEMBLY LINES, as mentioned earlier.

Although some of the material presented will be somewhat redundant, its repetition is preferred for the continuity of the text. As mentioned earlier, you may wish to read other articles in the series ASSEMBLY LINES or purchase ASSEMBLY LINES: THE BOOK for further information on machine language programming.

ASSEMBLY LINES - JANUARY, 1982

by Roger Wagner

One useful application of machine language programming is in the enhancement of your existing Applesoft programs. Some people are inclined to write all their programs in machine language, but I prefer on occasion to write what I call "hybrids", that is, programs which are a combination of Applesoft and machine language. In this way, particular functions can be done by the operating system best suited to the particular task.

If I had to write a short program to store 10 names, it would be best to write a short and simple program in Applesoft:

```
10  FOR I = 1 TO 10
20  INPUT N$(I)
30  NEXT I
```

This is much simpler than the equivalent program in machine language. In cases where neither speed nor program size is a concern, Applesoft is a completely acceptable solution.

However, if I had to SORT 1000 names, speed would become a concern, and it would be worth considering whether the job could best be done in machine language.

If you have ever done a ´CALL´ in one of your BASIC programs, then you have already combined Applesoft with machine code. For example:

```
10 HOME
20 PRINT "THIS IS A TEST"
30 PRINT "THIS IS STILL A TEST"
40 GET A$
50 HTAB 1: VTAB 5: CALL-958
```

In this program, a line of text is printed on the screen. After pressing a key, all text on the screen after the first word "THIS" is cleared.

Now although it would be possible to accomplish the same effect in Applesoft by perhaps printing many blank lines, it would not be as fast or as efficient code-wise as the CALL -958.

In executing the above program, the Applesoft interpreter goes along carrying out your instructions until it reaches the 'CALL' statement. At that point a JSR is done to the address indicated by the CALL. When the final RTS is encountered, control returns to the BASIC program. In between, however, you can do anything you'd like!

Calling routines is hardly complicated enough to warrant an entire article on the subject. The real question is, how do you pass data back and forth between the two programs, and how can the problem of handling that data be made easier for the machine language program?

SIMPLE INTERFACING

The easiest way to pass data to a machine language routine is to simply POKE the appropriate values into unused memory locations, and then retrieve them when you get to your machine language routine. To illustrate this, let's resurrect the tone routine from the May, 1981 issue of Softalk.

To use this, assemble the code, and place the final object code at $300. Then enter the accompanying Applesoft program.

```
 1  **********************
 2  * SOUND ROUTINE #3A *
 3  **********************
 4  *
 5  *
 6   OBJ $300
 7   ORG $300
 8  *
 9  PITCH EQU $06
10  DURATION EQU $07
11  SPKR EQU $C030
12  *
13  BEGIN LDX DURATION
14  LOOP LDY PITCH
15   LDA SPKR
16  DELAY DEY
17   BNE DELAY
18  DRTN DEX
19   BNE LOOP
20  EXIT RTS
```

From the Monitor, this will appear as:

```
*300L

0300-   A6 07       LDX   $07
0302-   A4 06       LDY   $06
0304-   AD 30 C0    LDA   $C030
0307-   88          DEY
0308-   D0 FD       BNE   $0307
030A-   CA          DEX
030B-   D0 F5       BNE   $0302
030D-   60          RTS
```

This Applesoft program is used to call it:

```
10 INPUT "PITCH, DURATION? ";P,D
20 POKE 6,P: POKE 7,D
30 CALL 768
40 PRINT
50 GOTO 10
```

The Applesoft program works by first requesting values for the pitch and duration of the tone from the user. These values are then POKE'd into locations 6 and 7 and the tone routine called. The tone routine uses these values to produce the desired sound, and then returns to the calling program for another round.

This technique works fine for limited applications. Having to POKE all the desired parameters into various corners of memory is not flexible, and strings are nearly impossible. There must be an alternative.

If you've been following this series for long you've no doubt figured out by now that I'm a great believer in using routines already present in the Apple where possible, to accomplish a particular task. Since routines already exist in Applesoft for processing variables directly, why not use them?

To answer this, I must take a brief detour to outline how Applesoft actually "runs" a program.

Consider this simple program:

```
10 HOME: PRINT "HELLO"
20 END
```

After entering this into the computer, typing "LIST" should reproduce the listing given here. An interesting question arises: "How does the computer actually store, and then later execute this program?".

To answer that, we'll have to go to the Monitor and examine the program data directly.

The first question to answer is, exactly where in the computer is the program stored? This can be found by entering the Monitor and typing in:

```
67 68 AF B0  (and pressing RETURN)
```

The computer should respond with:

```
67- 01
68- 08
AF- 18
B0- 08
```

The first pair of numbers is the pointer for the program beginning, bytes reversed of course. They indicate that the program starts at $801. The second pair is the program end pointer, and they show it ends at $818. Using this information, let's examine the program data by typing in:

```
801L
```

You should get:

```
*801L

0801-   10 08        BPL   $080B
0803-   0A           ASL
0804-   00           BRK
0805-   97           ???
0806-   3A           ???
0807-   BA           TSX
0808-   22           ???
0809-   48           PHA
080A-   45 4C        EOR   $4C
080C-   4C 4F 22     JMP   $224F
080F-   00           BRK
0810-   16 08        ASL   $08,X
0812-   14           ???
0813-   00           BRK
0814-   80           ???
0815-   00           BRK
0816-   00           BRK
0817-   00           BRK
0818-   F9 A2 00     SBC   $00A2,Y
081B-   86 FE        STX   $FE
```

This is obviously not directly executable code. Now type in:

```
801.818
```

This will give:

```
*801.818

0801- 10 08 0A 00 97 3A BA
0808- 22 48 45 4C 4C 4F 22 00
0810- 16 08 14 00 80 00 00 00
0818- 8C
```

To understand this, let's break it down one section at a time. When the Apple stores a line of BASIC, it encodes each keyword as a single byte "token". Thus the word "PRINT" is stored as a $BA. This does wonders for conserving space. In addition, there is some basic overhead associated with "packaging" the line, namely a byte at the end to signify the end of the line, and a few bytes at the beginning of each line to hold information related to the length of the line, and also the line number itself.

To be more specific:

```
0801- 10 08 0A 00 97 3A BA
0808- 22 48 45 4C 4C 4F 22 00
0810- 16 08 14 00 80 00 00 00
0818- 8C
```

The first two bytes of every line of an Applesoft program are an "index" to the address of the beginning of the next line. At $801,802 we find the address $810 (bytes reversed). This is where line 20 starts. At $810 we find the address $816. This is where the next line would start if there was one. The double ´00´ at $816 tells Applesoft that this is the end of the BASIC listing. It is important to realize that the ´00 00´ end of the Applesoft program usually corresponds to the contents of $AF,B0, BUT NOT ALWAYS. It is possible to hide machine language code between the end of the line data and the actual end as indicated by $AF,B0, but more on that later.

The next information within a line is the line number itself:

```
0801- 10 08 0A 00 97 3A BA
0808- 22 48 45 4C 4C 4F 22 00
0810- 16 08 14 00 80 00 00 00
0818- 8C
```

The ´0A 00´ is the two byte form of the number ´10´, the line number of the first line of the Applesoft program. Likewise, the ´14 00´ is the data for the line number ´20´. The bytes are again reversed. After these four bytes, we see the actual tokens for each line.

```
0801- 10 08 0A 00 97 3A BA
0808- 22 48 45 4C 4C 4F 22 00
0810- 16 08 14 00 80 00 00 00
0818- 8C
```

All bytes with a value of $80 or greater are Applesoft keywords in token form. Bytes less than $80 represent normal ASCII data (letters of the alphabet, etc.). Examining the data here we see a $97 followed by $3A. $97 is the token for 'HOME', and $3A the colon. Next, $BA is the token for 'PRINT'. This is followed by the quote ($22) and the text for HELLO (48 45 4C 4C 4F) and the closing quote ($22). Last of all, the '00' indicates the end of the line.

In line number 20, the $80 is the token for 'END'. As before, the line is terminated with a '00'.

When a program is executed, the interpreter scans through the data. Each time it encounters a token, such as the PRINT token, it looks up the value in a table to see what action should be taken. In the case of PRINT, this would be to output the characters following the token, namely "HELLO".

This constant translation is the reason for the use of the term "interpreter" for Applesoft BASIC.

Machine code on the other hand is directly executable by the 6502 microprocessor and is hence much faster since no table "lookups" are required.

In Applesoft, a SYNTAX ERROR is generated whenever a series of tokens is encountered that is not consistent with what the interpreter expects to find.

PASSING VARIABLES

So, back to the point of all this. The key to passing variables to your own machine language routines is to work with Applesoft in terms of routines already present in the machine. One of the simplest methods was described in the October, 1981 issue of Softalk, wherein a given variable is the very first one defined in your program (see the input routine). This is ok, but rather restrictive. A better way is to name the variable you're dealing with right in the CALL statement.

The important points here are the two components of the Applesoft interpreter: the TXTPTR and CHRGET (and related routines).

TXTPTR is the two byte pointer ($B8, B9) that points to the next token to be analyzed. CHRGET ($B1) is a very short routine that actually resides on the zero page which will read a given token into the Accumulator. In addition to occasionally being called directly, many other routines use CHRGET to process a string of data in an Applesoft program line.

Here then is the revised tone routine:

```
 1  **********************
 2  * SOUND ROUTINE #3B *
 3  **********************
 4  *
 5  *
 6   OBJ $300
 7   ORG $300
 8  *
 9  PITCH EQU $06
10  DURATION EQU $07
11  SPKR EQU $C030
12  *
13  COMBYTE EQU $E74C
14  *
15  ENTRY JSR COMBYTE
16   STX PITCH
17   JSR COMBYTE
18   STX DURATION
19  *
20  BEGIN LDX DURATION
21  LOOP LDY PITCH
22   LDA SPKR
23  DELAY DEY
24   BNE DELAY
25  DRTN DEX
26   BNE LOOP
27  EXIT RTS
```

This would list from the Monitor as:

```
*300L

0300-   20 4C E7    JSR   $E74C
0303-   86 06       STX   $06
0305-   20 4C E7    JSR   $E74C
0308-   86 07       STX   $07
030A-   A6 07       LDX   $07
030C-   A4 06       LDY   $06
030E-   AD 30 C0    LDA   $C030
0311-   88          DEY
0312-   D0 FD       BNE   $0311
0314-   CA          DEX
0315-   D0 F5       BNE   $030C
0317-   60          RTS
```

The Applesoft calling program would then be revised to read:

```
10 INPUT "PITCH, DURATION? ";P,D
20 CALL 768,P,D
30 PRINT
40 GOTO 10
```

This is a much more elegant way of passing the values and also requires no miscellaneous memory locations as such (although for purposes of simplicity the tone routine itself still uses the same zero page locations.)

The secret to the new technique is the use of the routine COMBYTE ($E74C). This is an Applesoft routine which checks for a comma and then returns a value between $00 and $FF (0-255) in the X-Register.

It is normally used for evaluating POKEs, HCOLOR=, etc., but does the job very nicely here. It also leaves TXTPTR pointing to the end of the line (or a colon if there was one) by using CHRGET to advance TXTPTR appropriate to the number of characters following each comma. Not also that any legal expression such as ´(X-5)/2´ can be used to pass the data.

To verify the importance of managing TXTPTR, try putting a simple RTS ($60) at $300. Calling this you will get a SYNTAX ERROR, since upon return, Applesoft´s TXTPTR will be on the first comma, and the phrase ",P,D" is not a legal Applesoft expression.

Now what about two-byte (values greater than 256) quantities? To do this, a number of other routines are used. For example, this routine will do the equivalent of a two-byte pointer POKE. Suppose for instance you wanted to store the bytes for the address $9600 at locations $1000, 1001. Normally in Applesoft you would do it like this:

```
.
.
.
50 POKE 4096,0: POKE 4097,150
.
.
.
```

Where 4096 and 4097 are the decimal equivalents of $1000 and $1001, and 0 and 150 are the low and high order bytes for the address $9600 ($96 = 150, $00 = 0).

A more convenient approach might be like this:

```
.
.
.
50 CALL 768, 4096, 38400
.
.
.
```

or perhaps:

```
.
.
.
50 CALL 768, A, V
.
.
.
```

The routine for this would be:

```
 1   **************************
 2   * POINTER SET UP ROUTINE *
 3   **************************
 4   *
 5   *
 6    OBJ $300
 7    ORG $300
 8   *
 9   CHKCOM EQU $DEBE
10   FRMNUM EQU $DD67
11   GETADR EQU $E752
12   LINNUM EQU $50 ; ($50,51)
13   *
14   PTR EQU $3C
15   *
16   ENTRY JSR CHKCOM
17    JSR FRMNUM ; EVAL FORMULA
18    JSR GETADR ; PUT FAC INTO LINNUM
19    LDA LINNUM
20    STA PTR
21    LDA LINNUM+1
22    STA PTR+1
23   *
24    JSR CHKCOM
25    JSR FRMNUM
26    JSR GETADR
27   *
28    LDY #$00
29    LDA LINNUM
30    STA (PTR),Y
31    INY
32    LDA LINNUM+1
33    STA (PTR),Y
34   *
35   DONE RTS
```

Which will list from the Monitor as:

```
*300L

0300-   20 BE DE    JSR   $DEBE
0303-   20 67 DD    JSR   $DD67
0306-   20 52 E7    JSR   $E752
0309-   A5 50       LDA   $50
030B-   85 3C       STA   $3C
030D-   A5 51       LDA   $51
030F-   85 3D       STA   $3D
```

```
0311-   20 BE DE    JSR   $DEBE
0314-   20 67 DD    JSR   $DD67
0317-   20 52 E7    JSR   $E752
031A-   A0 00       LDY   #$00
031C-   A5 50       LDA   $50
031E-   91 3C       STA   ($3C),Y
0320-   C8          INY
0321-   A5 51       LDA   $51
0323-   91 3C       STA   ($3C),Y
0325-   60          RTS
```

The special items in this routine include CHKCOM, a syntax checking routine that serves two purposes. First it verifies that a comma follows the CALL address, and secondly it advances TXTPTR to point to the first byte of the expression immediately following the comma. If a comma is not found, a SYNTAX ERROR is generated.

FRMNUM is a routine that evaluates any expression and puts the real floating point number result into Applesoft's "Floating Point Accumulator", "FAC" as it is usually called. This 6 byte psuedo register ($97-9C) is used to hold the floating point representation of a number. It includes such nifties as the exponential magnitude of the number and the equivalent of the digits of the logarithm of the number stored.

At this stage it would be a lot of work to deal with the number in its current form, so the next step is used to convert it into a two byte integer.

GETADR does this by putting the two byte result into LINNUM, LINNUM+1 ($50,51).

Even if this is not exactly an in-depth explanation of all the most precise details of the operation, the bottom line is that the three JSR's (CHKCOM, FRMNUM, and GETADR) will always end up with the low and high order bytes of whatever expression follows a comma in LINNUM and LINNUM+1.

These simple subroutines should be quite adequate for many applications. Next month, however, I'll explain passing strings, some of the various other routines available, and how to pass data back to the calling Applesoft program.

ASSEMBLY LINES - FEBRUARY, 1982

Last month, we began a discussion of how to pass variables back and forth between Applesoft and machine language programs. This month we'll complete the discussion with more information on how all types of variables are handled, and how data can also be passed back to the calling Applesoft program.

APPLESOFT VARIABLES

There are six types of variables in Applesoft BASIC. These consist of REAL, INTEGER, and STRING variables, and their array counterparts. To fully understand how to use these variables, we must first take a moment to examine the differences in each, and how the variables are actually stored in the computer.

Real variables are number values between 10^{38} and -10^{38}, that is to say very large positive and negative numbers. In addition, the values need not be whole numbers; a value such as 1.25 is allowed. Integer variables on the other hand are limited in magnitude to the range of +32768 to -32767. They are also limited to whole number values, i.e. 1,2,3,etc. Values such as 1.25 are not allowed.

Real variables are indicated in BASIC by an alphabetic character (A-Z) followed by a letter or number (A-Z,0-9). Any characters after the first two are ignored when Applesoft looks up the value for the variable. Integer variables are similar, but the name is suffixed by a percent sign (%). Thus 'A' would represent the Real variable, whereas 'A%' would represent an Integer variable.

When passing data such as a memory address or a single byte value to put in memory, Integer variables would be quite adequate and additionally, require no conversion in the machine language routine. However, it is generally more convenient to the BASIC programmer, not to have to put the '%' sign in the variable name, and to instead convert the value using the Applesoft routine "FRMNUM" ($DD67) as described in the last issue. For the record though, I will present an example shortly on how to retrieve an Integer variable from a calling BASIC program.

String variables consist of a series of any legal ASCII characters, with a maximum length of 255 characters. Strings are indicated by a '$' suffix to the variable name.

Any of these variables may be present either singly or in an ARRAY. Arrays are groupings of variables which use a common name, and then a delimiting SUBSCRIPT to identify each individual element. Array variables are indicated by a pair of parenthesis following the variable name, between which a number or expression may be used to specify the desired element.

Although I assume you are already somewhat familiar with the general points mentioned so far, I bring them up not so much to teach you about Applesoft variable types as such, but rather to set the stage for what is to follow, namely how each of these variable types is stored within the memory of the Apple computer.

MEMORY MAPS

Quite some time ago I presented a graphic representation of the memory usage of the computer. I would like to revive the topic in the interest of our current subject.

A memory map is used to show the relative placement of data within the available memory locations within the computer. Recall that there are a total of 65,536 locations available, which we identify with hexadecimal addresses of $0000 to $FFFF.

The chart below shows how a normal Apple would be shown, with DOS booted, and an arbitrary Applesoft program in memory.

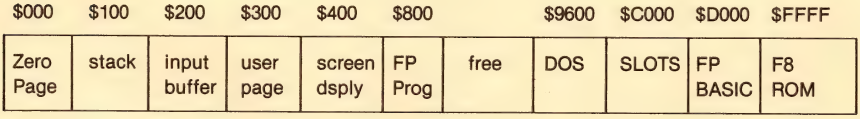


fig. 1

In previous articles, the areas shown have been desribed in varying degrees of detail. You'll recall that the area from $C000 to $CFFF is reserved for the interface card addressing, and that Applesoft BASIC is stored in ROM, beginning at $D000. The Monitor ROM begins at $F800.

A normal Applesoft program starts at $800, with the highest available address usually being $9600, which is identified with the lower boundary of the Disk Operating System (DOS).

The area from $300 to $3CF is availabe for user machine language programs. $3D0 to $3FF is reserved for Apple system vectors, such as the DOS entry vectors. Zero page, the stack, and the input buffer have also been discussed in some detail.

Since our main concern is in the area of Applesoft variables, let's consider a revised map, emphasizing Applesoft programs:

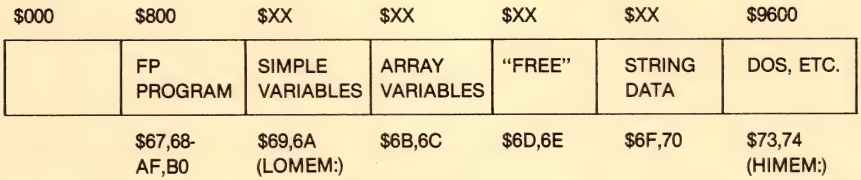


fig. 2

Fig. 2 shows that when an Applesoft program is run, simple (non-array) variables are placed immediately after the end of the BASIC program, followed by the array variables. Because the data for each string variables is always changing in lengt , string data is stored dynamic-ally at the top of memory, working down. The space in between these converging areas is the so-called "free space" of the system.

HIMEM: and LOMEM: are used by the BASIC programmer to set the upper and lower bounds of variable storage. If not specifically declared within the program, these default to the bottom of DOS and the end of the Applesoft program, respectively. They DO NOT, however, always have to be restricted to these locations. It is possible to move LOMEM: up, or HIMEM: down, so as to set aside a portion of memory in the computer which will not be affected by the running program. This is done for one or both of two reasons. First, to protect either or both of the HI-RES display pages from variable table encroachment, or second, to provide a protected area for a user's machine language program.

Now that we know where the information for each variable is stored in the computer, let's examine the format of the information for each variable. Within the areas indicated, a variable table is constructed which contains the name of the given variable, and its value if the variable is a real or integer. If the variable is a string, a pointer is stored which indicates exactly where at the top of memory the string is stored, and its corresponding length (0-255 characters).

Fig. 3 summarizes the details of the format for these tables:

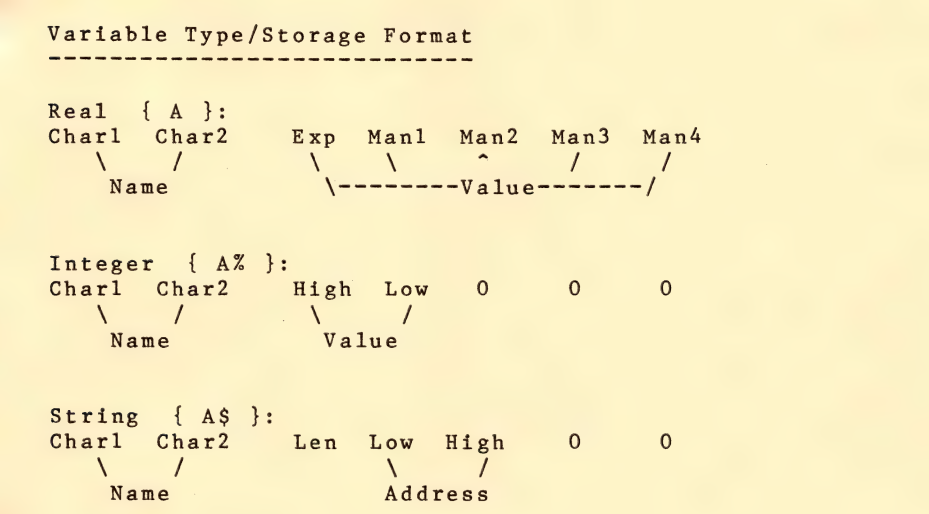


Fig. 3

Each time a variable is first encountered in a running Applesoft program, an entry in the variable table is made for it. For simple variables, Applesoft looks to the pointer at $6B,6C to see where the end of the current simple variable table is. It then opens up 7 bytes for the new variable and puts a block of data similar to that shown in fig. 3, as is appropriate to the type of variable defined.

Real variables store the value in a logarithmic form, where each value is indicated by the exponent and four mantissas. Integer variables require only the high and low order bytes of the value be stored. The remaining three positions are unused, with dummy '0' values placed in the table. It is important to note here that for integer variables, the two byte representation of the value is reversed from what we would normally expect. That is to say, the high order byte is placed first, followed by the low order byte.

For strings only three bytes of information are required, namely the length and address data mentioned earlier. Again the last two positions are filled with dummy '0's.

It should be evident from this table that the same amount of memory is allocated for all simple variable types, i.e. there is no advantage in specifying integer variables vs. reals to save memory. This will not be the case with arrays.

Notice that there are two distinct parts to each 7 byte variable entry. The first two bytes define the name, where incidentally, the high order byte is used in each character to indicate which of the three variable types (real, integer or string) that entry corresponds to. The last five bytes make up the actual data for each variable, and consists of either the required numeric information, or if it's a string, the length and address information.

The reason I mention this distinction is that in examining arrays, we notice that it is this five byte block that gets repeated a large number of times, depending on the total number of elements in the array.

For arrays a much larger table needs to be constructed, and this is created starting at the address indicated by $6B,6C. Whenever a new array is defined, the pointer at $6D,6E is examined to determine the end of the current array table, and a new entry made, according to the format shown in Fig. 4.

In this format, the entry is given a "header" that gives the variable name, followed by an offset value used to determine the address of the next array entry, if one is present. The offset is encoded in the usual two byte manner. Following the offset is is a byte indicating the number of dimensions in the array, after which is then listed a byte for each dimension stating its size. Although not shown in the diagram, each size indicator is a two byte pair, although in this case the high byte is always given first.

Immediately after the header is found the actual data blocks, each block consisting of 5, 2 or 3 bytes per array element, depending on which variable type is involved. Note that in this case, integer variable arrays do take much less memory than an equivalent real array.

Variable Type/Storage Format

Real Arrays: A(d1,d2,...,dn)

```
Char1  Char2  OSL  OSH   Num   dn ... d1  [5 Byte blocks]
  \     /      \    /     ^      \    /     \           /
    Name        Offset  # of Dims  Sizes      Actual Data
```

Integer Arrays: A%(d1,d2,...,dn)

```
Char1  Char2  OSL  OSH   Num   dn ... d1  [2 Byte blocks]
  \     /      \    /     ^      \    /     \           /
    Name        Offset  # of Dims  Sizes      Actual Data
```

String Arrays: A$(d1,d2,...,dn)

```
Char1  Char2  OSL  OSH   Num   dn ... d1  [3 Byte blocks]
  \     /      \    /     ^      \    /     \           /
    Name        Offset  # of Dims  Sizes      Actual Data
```

Fig. 4

As an example, if you were to dimension an array with this statement: DIM A$(10,10)

The header block would look like this:

```
$41  $80  $74  $01  $02  $00 $0B  $00 $0B [3 Byte blocks]
 \    /    \    /    ^      \        /      \         /
  Name      Offset  Dims.       Sizes          Actual Data
```

Fig. 5

Where $41,80 are the ASCII values for an ´A´ followed by a null. High bit is off in the first character, on in the second, indicating a string. The next array variable would be found at the address of the first name character plue $174. There are two dimensions to the array, indicated by the $02. The $00 $0B indicates ELEVEN elements in each dimension of the array. This should not be surprising when you recall that ´10´ plus the ´0´th position makes eleven elements.

Following this header we would find 121 three-byte blocks, each indicating the length and address of a string array element, if present. (11 x 11 = 121; (121 * 3) + 9 (header) = 372 = $174!)

PASSING VARIABLES FROM APPLESOFT TO MACHINE LANGUAGE

At this point you may well think that we have strayed very far from the topic of machine language programming, and have become overly involved with the structure of Applesoft. Upon a little reflection however, it should become apparent that we must have some familiarity with how these variables are stored if we are to successfully interact with them.

In either reading or creating Applesoft variables, clearly we must effectively handle each component of the data. We must be able to identify the name and location of the variable we are interested in, and to also modify that information if necessary.

The temptation at this point might be to take this new found knowledge and write our own routines to accomplish the needed operations independent of Applesoft, but I assure your such an undertaking would be quite unnecessary, not to mention likely to have you mindlessly babbling to yourself in no time.

Fortunately Applesoft already contains all of the necessary routines to do almost anything we wish. The main trick will be to properly identify and use the appropriate ones.

Last month I made use of a few of these to accomplish a certain degree of flexibility in passing numeric data to a machine language routine. Let's complete the study by formalizing the possible operations.

The first general category is passing data to a routine. We can pass any of six variable types. To minimize the confusion, let us establish a fairly simple goal: to successfully pass the data, and prove so by storing the data in a non-Applesoft location.

INTEGER VARIABLES

First for integer variables. The calling Applesoft program looks like this:

```
10  A% = 258
20  CALL 768,A%
30  PRINT PEEK(896),PEEK(897)
40  REM 896,897 = $380,381
50  END
```

The machine language routine should be be assembled from this listing:

```
                    1    *****************
                    2    * INT VARIABLE  *
                    3    *    READER     *
                    4    *    2/1/82     *
                    5    *****************
                    6    *
                    7    *
                    8              OBJ  $300
                    9              ORG  $300
                   10    *
                   11    CHKCOM    EQU  $DEBE
                   12    PTRGET    EQU  $DFE3
                   13    VARPNT    EQU  $83
                   14    MOVFM     EQU  $EAF9
                   15    CHKNUM    EQU  $DD6A
                   16    DATA      EQU  $380
                   17    *
0300: 20 BE DE     18    ENTRY     JSR  CHKCOM ; CHK SYNTAX
0303: 20 E3 DF     19              JSR  PTRGET ; FIND VARIABLE
                   20    * Y,A = ADDR OF VALUE
0306: 20 F9 EA     21              JSR  MOVFM  ; MOVE VAL-> FAC
0309: 20 6A DD     22              JSR  CHKNUM ; FAC = NUM?
030C: A0 00        23              LDY  #$00
030E: B1 83        24              LDA  (VARPNT),Y
0310: 8D 81 03     25              STA  DATA+1
0313: C8           26              INY
0314: B1 83        27              LDA  (VARPNT),Y
0316: 8D 80 03     28              STA  DATA
                   29    *
                   30    * NOTE! HIGH BYTE FIRST!
                   31    *
0319: 60           32    DONE      RTS
```

In this routine, CHKCOM ($DEBE = "Check for Comma") is used to make sure the syntax is correct (ie. a comma), and to advance TXTPTR ($B8 = "Text Pointer") to the first byte of the variable name being evaluated. Refer to last month's issue for the original discussion on these two routines.

PTRGET ($DFE3 = "Pointer Get") is now called, which is a subroutine which reads a variable name in and then locates it in the variable table. As an added bonus, if the variable named does not currently exist in the table, it will create an entry for it. This applies to variables of all six types. After returning from PTRGET, the address of the value for the variable is held in the Y-Register and the Accumulator (low byte, high byte). This thus indicates the location in memory of the 2 to 5 byte data block discussed earlier. The data in the Y-Register and Accumulator is also duplicated in VARPNT, VARPNT+1 ($83,84 = "Variable Pointer"), which will be used later in the program.

At this stage it would be a simple matter to use indirect addressing to retrieve the two bytes, but a little more effort will result in a much more thorough routine. It is possible that the user might have called the routine with an improper variable type following the CALL statement, such as a string. This can be checked for by the next two program steps.

MOVFM ($EAF9 = "Move to FAC from Memory") will move whatever data is pointed to by the Y-Register and the Accumulator into the Floating Point Accumulator ($F9-A2 = "FAC"). The contents can then be checked for variable type by the call to CHKNUM ($DD6A = "Check Number"). The presence of a string here would yield a TYPE MISMATCH ERROR. Unfortunately, it is not particularly easy to test for a real variable having been mistakenly used here.

Presuming no error occurs, we will now make use of the data saved in VARPNT (since the Y-Register and Accumulator have been no doubt altered by MOVFM and CHKSTR) to actually retrieve the two byte value passed. The indirect addressing mode is now used to move the variable data into our two DATA bytes. The address of $380,381 was in this case arbitrarily chosen for the example.

It is important to note that special care is used in lines 25 and 28, since integer variables store the two data bytes high order byte first, as mentioned earlier. This is opposite to the normal 6502 convention.

This routine will work equally well for retrieving data from both simple integer variables and integer array variables.

When you run this example, the numbers "2" and "1" should be printed out, these being the low and high order bytes of the number passed to the routine (258 = $102).

REAL VARIABLES

Once in machine language, the handling of floating point numbers, such as represented by real variables is somewhat involved. Additionally, the majority of the time you will be concerned only with passing an integer between 0 and 65535. Therefore, we will consider here how to use a real variable to pass a number in this range to a given subroutine.

This revision of our first Applesoft program will do the trick:

```
10  A = 258
20  CALL 768,A
30  PRINT PEEK(896),PEEK(897)
40  REM 896,897 = $380,381
50  END
```

The assembly language program for this is:

```
1     ******************
2     * REAL VARIABLE *
3     *    READER     *
4     *    2/1/82     *
5     ******************
6     *
7     *
8               OBJ  $300
9               ORG  $300
10    *
```

```
                    11   CHKCOM   EQU  $DEBE
                    12   FRMNUM   EQU  $DD67
                    13   GETADR   EQU  $E752
                    14   LINNUM   EQU  $50
                    15   DATA     EQU  $380
                    16
0300: 20 BE DE      17   ENTRY    JSR  CHKCOM ; CHK SYNTAX
0303: 20 67 DD      18            JSR  FRMNUM ; EVALUATE NUM
0306: 20 52 E7      19            JSR  GETADR ; FAC -> INT
0309: A5 50         20            LDA  LINNUM
030B: 8D 80 03      21            STA  DATA
030E: A5 51         22            LDA  LINNUM+1
0310: 8D 81 03      23            STA  DATA+1
0313: 60            24   DONE     RTS
```

This is basically a repeat of last month's routine, with the results being put in DATA,DATA+1. The advantage of this routine is that not only is it shorter, but it will accept either integer or real variables (simple or array), and still do the string error check. This then is usually the preferred method.

STRING VARIABLES

The goal here will be to read some string data from the calling Applesoft program, and to then put it somewhere in memory, where it would presumably be available to other portions of the machine language program. To illustrate this, enter the following two programs:

```
10  A$ = "TEST"
20  CALL 768,A$
30  END
```

```
                    1    ********************
                    2    * STR$ VAR READER *
                    3    *     2/1/82       *
                    4    *   R. WAGNER      *
                    5    ********************
                    6    *
                    7    *
                    8              OBJ  $300
                    9              ORG  $300
                    10   *
```

```
                        11    CHKCOM    EQU  $DEBE
                        12    FRMEVL    EQU  $DD7B
                        13    CHKSTR    EQU  $DD6C
                        14    FACMO     EQU  $A0 ; FAC+4
                        15    FACLO     EQU  $A1 ; FAC+5
                        16    VARPNT    EQU  $83
                        17    DATA      EQU  $380
                        18    *
0300: 20 BE DE          19    ENTRY     JSR  CHKCOM ; CHK SYNTAX
0303: 20 7B DD          20              JSR  FRMEVL ; EVALUATE
                        21    * (FACMO,LO) -> DESCRIPTOR
0306: 20 6C DD          22              JSR  CHKSTR ; VAR = $?
                        23    *
0309: A0 00             24              LDY  #$00
030B: B1 A0             25              LDA  (FACMO),Y ; LEN OF $
030D: AA                26              TAX  ; SAVE LEN
030E: C8                27              INY  ; Y = 1
030F: B1 A0             28              LDA  (FACMO),Y ; ADR LO BYTE
0311: 85 83             29              STA  VARPNT
0313: C8                30              INY  ; Y = 2
0314: B1 A0             31              LDA  (FACMO),Y ; ADR HI BYTE
0316: 85 84             32              STA  VARPNT+1
0318: 8A                33              TXA  ; RETRIEVE LEN
0319: A8                34              TAY
                        35    *
031A: 88                36    LOOP      DEY
031B: B1 83             37              LDA  (VARPNT),Y ; GET CHR
031D: 99 80 03          38              STA  DATA,Y
0320: C0 00             39              CPY  #$00
0322: D0 F6             40              BNE  LOOP
                        41    *
0324: 60                42    DONE      RTS
```

After running the calling program, enter the Monitor, and list out the DATA region of memory with:

```
*380.383 <RETURN>
```

This should print out the following data:

```
0380- 54 45 53 54
```

This shows that the hex values for the characters "TEST" have been successfully transferred. Let's see how it was accomplished.

IF getting Formula Too Complex in above ADD A JSR to E653 just Before last RTS

E5FD

The routine starts off rather like the previous ones by using CHKCOM to make sure a comma was used after the CALL and to prepare TXTPTR for reading in the data. FRMEVL ($DD78 = "Formula Evaluation") is a very nice general purpose routine which takes in virtually any numeric or string expression or literal, and places the final result in FAC. It is related to FRMNUM, but is much more omnivorous. Upon returning from FRMEVL, FACMO and FACLO ($A0,A1 = "... sorry, couldn't find out where they got the name!...") hold the address of the string's "Descriptor", that is, the 3 byte group giving the length and address of the actual string data.

Our routine uses FACMO,FACLO in the indirect addressing mode to retrieve the first byte of the descriptor, which is the length of the string. This is put into the X-Register for temporary storage. Some people would prefer to push it onto the stack with a PHA command, but it's a matter of choice. Next the address of the string data is retrieved from the descriptor and put into VARPNT, which is assumed to be not in use at the time. Last of all we use the VARPNT pointer to move the data from it's location, indicated by VARPNT, to our DATA address. In experimenting notice that the area from $380 to $3CF is open, but starting at $3D0, the area is reserved for DOS. Entering very long strings in the example may lead to some problems. In your own programs it would be necessary to set aside a one page area ($100 = 256 bytes) to put the data, unless of course you can limit the length of the string before doing the CALL.

You may also wish to try variations in the Applesoft program by deleting line 10 and rewriting line 20 as:

```
20 CALL 768,"ABC" + "DEF"

or

20 CALL 768,LEFT$("ABCDEF")

or

10 A$(5,5) = "TEST"

20 CALL 768,A$(5,5)
```

PASSING DATA FROM MACHINE LANGUAGE TO APPLESOFT VARIABLES

The converse of the techniques we've discussed so far is actually fairly simple. The key to much of it is the PTRGET routine that was used earlier. Because this routine will even create a variable if it's not already present, we can simply more or less reverse the process of the previous routines to pass data back to a calling Applesoft program.

Again, I'll illustrate an example for each variable type.

INTEGER VARIABLES

The Applesoft program:

```
10 POKE 896,2: POKE 897,1
20 CALL 768,A%
30 PRINT A%
40 END
```

The machine subroutine to be called is:

```
1  ******************
2  * INT VARIABLE   *
3  *    SENDER      *
4  *    2/1/82      *
5  ******************
6  *
7  *
8             OBJ   $300
9             ORG   $300
10 *
11 CHKCOM     EQU   $DEBE
12 PTRGET     EQU   $DFE3
13 VARPNT     EQU   $83
14 MOVFM      EQU   $EAF9
15 CHKNUM     EQU   $DD6A
16 DATA       EQU   $380
17 *
```

```
                    18 ENTRY    JSR  CHKCOM  ; CHK SYNTAX
                    19          JSR  PTRGET  ; FIND VARIABLE
                    20   * Y,A = ADDR OF VALUE
0306: 20 F9 EA      21          JSR  MOVFM ; MOVE VAL -> FAC
0309: 20 6A DD      22          JSR  CHKNUM ; FAC = NUM?
030C: A0 00         23          LDY  #$00
030E: AD 81 03      24          LDA  DATA+1
0311: 91 83         25          STA  (VARPNT),Y
0313: C8            26          INY
0314: AD 80 03      27          LDA  DATA
0317: 91 83         28          STA  (VARPNT),Y
                    29   *
                    30   * NOTE! HIGH BYTE FIRST!
                    31   *
0319: 60            32   DONE   RTS
```

This program is a rather trivial exercise in that all that need be done is to reverse the operands of lines 24,25 and 27,28 from the first Integer Reader program. Again, the only caution is to make sure the bytes are transferred in the proper order, since integer data is reversed.

REAL VARIABLES

Real variables require the introduction of a few new routines. The same Applesoft calling program is used with only a minor modification.

```
10 POKE 896,2: POKE 897,1
20 CALL 768,A
30 PRINT A
40 END
```

The subroutine is entered as:

```
                     1  ****************
                     2  * REAL VARIABLE *
                     3  *    SENDER     *
                     4  *    2/1/82     *
                     5  ****************
                     6  *
                     7  *
                     8          OBJ  $300
                     9          ORG  $300
                    10  *
```

```
                    11 CHKCOM    EQU  $DEBE
                    12 PTRGET    EQU  $DFE3
                    13 CHKNUM    EQU  $DD6A
                    14 GIVAYF    EQU  $E2F2
                    15 MOVMF     EQU  $EB2B
                    16 DATA      EQU  $380
                    17 *
0300: 20 BE DE      18 ENTRY     JSR  CHKCOM ; CHK SYNTAX
0303: AC 80 03      19           LDY  DATA
0306: AD 81 03      20           LDA  DATA+1
0309: 20 F2 E2      21           JSR  GIVAYF ; DATA -> FAC
030C: 20 E3 DF      22           JSR  PTRGET ; LOCATE VAR
030F: 20 6A DD      23           JSR  CHKNUM ; VAR = #?
                    24 * Y,A = ADDR OF VAR DATA
0312: AA            25           TAX
0313: 20 2B EB      26           JSR  MOVMF ; PUT FAC->MEMORY
0316: 60            27 DONE      RTS
```

The technique here is to use the routine GIVAYF ($E2F2 = "Give Acc & Y-Reg to FAC") to put the two bytes of our integer number into the FAC. GIVAYF requires that The Accumulator and Y-Register be loaded with the high and low order bytes, respectively, for the integer number to be transferred. As an added bonus, the number may even be "signed", that is positive or negative. Note that by using GIVAYF, numbers greater than 32767 will be returned as negative values. Signed binary numbers were briefly touched upon in an earlier issue, and are covered in greater detail in the book version of this series.

Lines 19 & 20 load the appropriate registers, and after calling GIVAYF and PTRGET are used to determine the name of the variable to use in returning the data. CHKNUM is then called to make sure it's a numeric variable (as opposed to a string). Recall that after returning from PTRGET, the Y-Register and Accumulator will hold the low and high order bytes of the address of the data for the new variable digested by PTRGET.

MOVMF ($EB2B = "Move to Memory from FAC") is the routine we'll use to complete the process. It requires that the YRegister and X-Register be loaded with the address of the memory location to which the contents of the FAC will be moved. Since PTRGET has just determined that for us, the only hitch is that PTRGET left the high order byte in the Accumulator instead of the X-Register as we require. A simple TAX solves that problem, and the routine is concluded with the call to MOVMF and an RTS.

High Byte in Y
Lo Byte in X

PROGRAMMING TIP: Whenever a routine ends with a 'JSR' to another routine, immediately followed by the ending RTS of the main routine, the listing can be shortened one byte by changing the last JSR to a JMP. When the RTS in the last called subroutine is encountered, the RTS will cause an exit from the main routine instead. An example of this would be to rewrite the end of the program just listed as:

```
                .
                .
                .
030F: 20 6A DD 23            JSR  CHKNUM ; VAR = #?
               24 * Y,A = ADDR OF VAR DATA
0312: AA       25            TAX
0313: 4C 2B EB 26 DONE       JMP  MOVMF ; PUT FAC ->
                                          MEMORY AND
                                          RETURN!
```

STRING VARIABLES

String variables are not much different, but will require a slightly clumsy calling Applesoft program to demonstrate. Line 10 is a series of 'POKE's which will put the ASCII data for the string "TEST" into memory at our usual DATA ($380) location. Additionally, a delimiter will be placed at the end of the string so that the routines we will be calling can determine the string's length. Use of a delimiter is more practical especially in situations where you don't know the length of an incoming string until the carriage return or other delimiter shows up. The Applesoft routine we'll be using will automatically determine the length by scanning the string for the delimiter.

```
10 POKE 896,84: POKE 897,69: POKE 898,83:
   POKE 899,84: POKE 900,0
20 REM "TEST" + NULL DELIMITER
30 CALL 768, A$
40 PRINT A$
50 END
```

The subroutine for this is:

```
                    1     ********************
                    2     * STR$ VAR SENDER *
                    3     *      2/1/82      *
                    4     *    R. WAGNER     *
                    5     ********************
                    6     *
                    7     *
                    8               OBJ   $300
                    9               ORG   $300
                    10    *
                    11    CHKCOM    EQU   $DEBE
                    12    PTRGET    EQU   $DFE3
                    13    CHKSTR    EQU   $DD6C
                    14    FORPNT    EQU   $85
                    15    MAK$      EQU   $E3E9
                    16    SAVD      EQU   $DA9A
                    17    DATA      EQU   $380
                    18    *
                    19    *
0300: 20 BE DE      20    ENTRY     JSR   CHKCOM ; SYNTAX?
0303: 20 E3 DF      21              JSR   PTRGET ; FIND VAR
0306: 20 6C DD      22              JSR   CHKSTR ; VAR = $?
0309: 85 85         23              STA   FORPNT
030B: 84 86         24              STY   FORPNT+1 ; ADR OF DESCR
030D: A9 80         25              LDA   #$80
030F: A0 03         26              LDY   #$03 ; A,Y = $380
0311: A2 00         27              LDX   #$00 ; DELIMITER = '00'
0313: 20 E9 E3      28              JSR   MAK$ ; DATA -> MEMORY
0316: 20 9A DA      29              JSR   SAVD ; VARPNT= NEW STRG
0319: 60            30    DONE      RTS
```

The new routines here are MAK$ ($E3E9 = "Make string") and SAVD ($DA9A = "Save Descriptor"). MAK$ requires that the Accumulator and the Y-Register hold the address (low, high) of the string to be scanned, and that the X-Register hold the value for the delimiting character. I have used '00' in this example, but another common variation would be to use a carriage return ($8D) or a comma ($2C). (Note that RETURN is almost always found in the input buffer with the high bit SET, ie. $8D vs. $0D).

After scanning for the delimiter, MAK$ moves the data up to the string storage area at the top of memory.

SAVD is a companion routine which will take whatever string descriptor is currently pointed to by FORPNT ($85,86 = "Formula Pointer(?)"), and match it to the data just moved by MAK$.

Looking at the listing, we can see that the only creative work that needs to be done is to move the contents of A and Y to FORPNT. The A, Y, and X registers are then prepared, as was just described, and the remaining calls done. Voila! Instant strings!

CLOSING STUFF

You'll notice that all of the routines handle arrays as well as simple variables. Additionally, certain more subtle points may become apparent as you study the listings. For example, each of the last three Applesoft listings was done without defining the returned variable prior to the CALL. This was to demonstrate that PTRGET does a very nice job of creating the variable for us. In addition, in each case, the data put into a variable, and then later retrieved at DATA (and vice versa) should be consistent, thus demonstrating the accuracy of the methods.

You may also wish to experiment with using formulas or string calculations after the CALL statement, to confirm that all legal Applesoft operations are acceptable.

CONCLUDING NOTES ON WRITING YOUR OWN

If you wish to write your own machine language routines for use with the Routine Machine then the first requirement is a good assembler (such as Southwestern Data Systems' MERLIN). Techniques of assembly language programming can be learned by studying the relevant articles in the computer magazines and the occasional helpful book. From then on you have to rely on your own ingenuity.

Routines for use with the Routine Machine should be relocatable (that is, executable at any location in memory). This simply means that you must not refer to any memory locations within your routine (such as with internal JSRs). References to external subroutines in the Apple Monitor ($F800-$FFFF), Applesoft ($D000-$F7FF) and DOS ($9D00-$BFFF on a 48K Apple) are OK. In fact, there are so many useful routines available in these three areas that most of your work has already been done for you, and it is simply a matter of learning what routines are available and how to use them. In the case of the routines in Applesoft, there is a wealth of information to be found in John Crossley's widely available article "Applesoft Internal Entry Points".

A Routine Machine routine should begin by reading the parameter list (if any) in the module invocation. Some information on this point has been given earlier in this manual in the sections concerned with passing parameters between your Applesoft program and invoked routines. Essentially your routine simply has to go through the bytes in the program text which follow the invocation name, using the same routines (PTRGET, CHKCOM, COMBYTE, FRMEVL, FRMNUM, COMBYTE, CHRGET, SYNCHR, etc.) that Applesoft itself uses when interpreting a program line. It is good practice to read the entire parameter list before proceeding further. However, if a syntax error (as defined either by Applesoft or by yourself) or some other error, such as a type mismatch error, occurs then the appropriate Applesoft error message can be generated immediately. If the syntax is correct, and the values of the parameters are acceptable, then the routine can proceed with its appointed task.

Option 8 in the Routine Machine allows you to see exactly what your module invocaton looks like when it has been tokenized and is sitting in memory. The bytes (if any) following the invocation name are what your routine will have to process when getting the data it needs.

When reading the parameter list it is essential to keep track of where TXTPTR ($B8,B9) is pointing. The routine CHRGOT (at $B7) will load the accumulator with whatever byte TXTPTR is pointing to, and CHRGET (at $B1) will advance TXTPTR to the next byte and load it into the accumulator. If the byte loaded is a colon ($3A) or an end-of-line token ($00) then both CHRGOT and CHRGET return with the zero flag set; this is how you can tell when you have reached the end of the parameter list.

There is one feature of the Interface Routine which is important to understand in this connection: If the first byte in the parameter list is a comma then the Interface Routine will gobble it before it passes control to your routine. In this case TXTPTR will be pointing to the byte immediately following the comma when control passes to your routine. If there is no parameter list following the invocation name, or if there is a parameter list but its first byte is not a comma, then when control passes to your routine TXTPTR will be pointing to the byte immediately following the second quote (") in the module invocation.

When debugging a machine language routine it is usually essential to know its precise location in memory. Thus it is better to test your routine at a fixed location in memory (e.g. $300 if it is short) BEFORE appending it to an Applesoft program as a Routine Machine module. If a relocatable routine functions properly at a fixed location then it will also function properly as a Routine Machine module.

The only complication here is that, if the parameter list that your routine will read has an initial comma (as is usually the case), then you must allow for the fact that (when your routine is installed as a Routine Machine module) this comma will be gobbled by the Interface Routine. It has been found in practice that when debugging a routine by CALLing it at a fixed location it is best to adopt the method of adding a JSR CHKCOM at the start to gobble the comma in the parameter list (but don't include this JSR CHKCOM in the final routine).

The Interface Routine is so designed that when your routine receives control it may find its own address in the zero page locations $5E,5F. This allows you to access data bytes (e.g. text) within your routine, which is normally not possible in a relocatable routine.

End your routine as usual with an RTS (or a JMP to an external routine which itself ends with an RTS). Control will then return to the Applesoft Interpreter, which will then proceed to execute the statement immediately following your module invocation.

APPLESOFT TOKEN CHART

DEC	HEX	CHR	DEC	HEX	CHR	DEC	HEX	CHR	LOWER CASE
0	0	^@	43	2B	+	85	55	U	
1	1	^A	44	2C	,	86	56	V	
2	2	^B	45	2D	-	87	57	W	
3	3	^C	46	2E	.	88	58	X	
4	4	^D	47	2F	/	89	59	Y	
5	5	^E	48	30	0	90	5A	Z	
6	6	^F	49	31	1	91	5B	[	
7	7	^G	50	32	2	92	5C	\	
8	8	^H	51	33	3	93	5D	]	
9	9	^I	52	34	4	94	5E	^	
10	A	^J	53	35	5	95	5F	_	
11	B	^K	54	36	6	96	60	Spc	(`)
12	C	^L	55	37	7	97	61	!	(a)
13	D	^M	56	38	8	98	62	"	(b)
14	E	^N	57	39	9	99	63	#	(c)
15	F	^O	58	3A	:	100	64	$	(d)
16	10	^P	59	3B	;	101	65	%	(e)
17	11	^Q	60	3C	<	102	66	&	(f)
18	12	^R	61	3D	=	103	67	'	(g)
19	13	^S	62	3E	>	104	68	(	(h)
20	14	^T	63	3F	?	105	69	)	(i)
21	15	^U	64	40	@	106	6A	*	(j)
22	16	^V	65	41	A	107	6B	+	(k)
23	17	^W	66	42	B	108	6C	,	(l)
24	18	^X	67	43	C	109	6D	_	(m)
25	19	^Y	68	44	D	110	6E	.	(n)
26	1A	^Z	69	45	E	111	6F	/	(o)
27	1B	Esc	70	46	F	112	70	0	(p)
28	1C	^\	71	47	G	113	71	1	(q)
29	1D	^]	72	48	H	114	72	2	(r)
30	1E	^^	73	49	I	115	73	3	(s)
31	1F		74	4A	J	116	74	4	(t)
32	20	Spc	75	4B	K	117	75	5	(u)
33	21	!	76	4C	L	118	76	6	(v)
34	22	"	77	4D	M	119	77	7	(w)
35	23	#	78	4E	N	120	78	8	(x)
36	24	$	79	4F	O	121	79	9	(y)
37	25	%	80	50	P	122	7A	:	(z)
38	26	&	81	51	Q	123	7B	;	({)
39	27	'	82	52	R	124	7C	<	(|)
40	28	(	83	53	S	125	7D	=	(})
41	29	)	84	54	T	126	7E	>	(~)
42	2A	*				127	7F	?	Rubout

^A = control-A, etc.

DEC	HEX	CHR	DEC	HEX	CHR	DEC	HEX	CHR
128	80	END	171	AB	GOTO	214	D6	FRE
129	81	FOR	172	AC	RUN	215	D7	SCRN(
130	82	NEXT	173	AD	IF	216	D8	PDL
131	83	DATA	174	AE	RESTORE	217	D9	POS
132	84	INPUT	175	AF	&	218	DA	SQR
133	85	DEL	176	B0	GOSUB	219	DB	RND
134	86	DIM	177	B1	RETURN	220	DC	LOG
135	87	READ	178	B2	REM	221	DD	EXP
136	88	GR	179	B3	STOP	222	DE	COS
137	89	TEXT	180	B4	ON	223	DF	SIN
138	8A	PR#	181	B5	WAIT	224	E0	TAN
139	8B	IN#	182	B6	LOAD	225	E1	ATN
140	8C	CALL	183	B7	SAVE	226	E2	PEEK
141	8D	PLOT	184	B8	DEF	227	E3	LEN
142	8E	HLIN	185	B9	POKE	228	E4	STR$
143	8F	VLIN	186	BA	PRINT	229	E5	VAL
144	90	HGR2	187	BB	CONT	230	E6	ASC
145	91	HGR	188	BC	LIST	231	E7	CHR$
146	92	HCOLOR=	189	BD	CLEAR	232	E8	LEFT$
147	93	HPLOT	190	BE	GET	233	E9	RIGHT$
148	94	DRAW	191	BF	NEW	234	EA	MID$
149	95	XDRAW	192	C0	TAB(	235	EB	
150	96	HTAB	193	C1	TO	236	EC	
151	97	HOME	194	C2	FN	237	ED	
152	98	ROT=	195	C3	SPC(	238	EE	
153	99	SCALE=	196	C4	THEN	239	EF	
154	9A	SHLOAD	197	C5	AT	240	F0	
155	9B	TRACE	198	C6	NOT	241	F1	
1561	9C	NOTRACE	199	C7	STEP	242	F2	
157	9D	NORMAL	200	C8	+	243	F3	
158	9E	INVERSE	201	C9	-	244	F4	
159	9F	FLASH	202	CA	*	245	F5	
160	A0	COLOR=	203	CB	/	246	F6	
161	A1	POP	204	CC	^	247	F7	
162	A2	VTAB	205	CD	AND	248	F8	
163	A3	HIMEM:	206	CE	OR	249	F9	
164	A4	LOMEM:	207	CF	>	250	FA	
165	A5	ON ERR	208	D0	=	251	FB	
166	A6	RESUME	209	D1	<	252	FC	
167	A7	RECALL	210	D2	SGN	253	FD	
168	A8	STORE	211	D3	INT	254	FE	
169	A9	SPEED=	212	D4	ABS	255	FF	
170	AA	LET	213	D5	USR			

>> APPENDIX D <<

SELECTED REFERENCES

This bibliography consists almost entirely of references to articles appearing during the period of January 1980 to September 1981 in the following publications:

* Micro
* Call A.P.P.L.E.
* Nibble
* Apple Assembly Line

The first three are well-known magazines, available in most computer stores. Apple Assembly Line is available from S-C Software, P.O. Box 280300, Dallas, Texas, 75228.

Most of these articles contain assembly language programs. Many of these require no modification, or only minor modification, to be usable with Routine Machine. If a routine can be used without modification, a short description of it is included with the reference.

The references are in order according to date of publication.

"& NOW, THE AMPERSAND", Anon., PEEKing at Call A.P.P.L.E., Vol. 1 (1978).

"& NOW, THE FURTHER ADVENTURES OF THE MYSTERIOUS AMPERSAND", Anon., PEEKing at Call A.P.P.L.E., Vol. 1 (1978).

"APPLE][MACHINE CODE RELOCATION", S. Wozniak, Wozpak][, Call A.P.P.L.E. (Nov 79).

"DOCUMENTATION: PROGRAMMER´S UTILITY PAK", R. Wagner, Southwestern Data Systems, 1979.

"APPLESOFT STRING SWAP", R. Wiggington, Call A.P.P.L.E., (Jan 80). Ampersand-invokable machine language routine for interchanging string variables.

"APPLE][PLUS FLOATING POINT UTILITY ROUTINES", H.L. Pruetz, Micro (Mar 80).

"APPLESOFT INTERNAL ENTRY POINTS", J. Crossley, Apple Orchard, (Mar/Apr 80). Reprinted in All About Applesoft, Call A.p.p.l.e. (1981).

"THE &LOMEM: UTILITY", N. Konzen, Apple Orchard (Mar/Apr 80).

"THE RETURN OF THE MYSTERIOUS MR. AMPERSAND", D. Lingwood, Call A.P.P.L.E. (May 80).

"APPLE STRINGS", R. Geiger, Creative Computing (May 80).

"TEXT OUTPUT ON THE APPLE][", R. Wagner, Call A.P.P.L.E. (Jun 80).

"ZOOM & SQUEEZE", G.B. Little, Micro (Jul 80).

"TYPES OF MEMORY MOVES", L. Reynolds, Call A.P.P.L.E. (Jul/Aug 80). Relocatable backwards memory move routine.

"HI-RES SCREEN SWITCH", W. Huntress, Call A.P.P.L.E. (Jul/Aug 80).

"AMPER-INTERPRETER", R.M. Mottola, Nibble (Aug/Sep 80).

"HI-RES SCREEN FUNCTION", K. Manly, Call A.P.P.L.E. (Oct 80).

"LINKING MACHINE LANGUAGE ROUTINES TO APPLESOFT PROGRAMS", Anon., Apple Orchard (Fall 80).

"GENERAL MESSAGE PRINTING ROUTINE", B. Sander-Cederlof, Apple Assembly Line (Oct 80).

"PRINT USING FOR APPLESOFT", G.A. Morris, Micro (Oct 80).

"HOW TO USE THE HOOKS", R. Williams, Micro (Nov 80).

"BASIC/MACHINE LANGUAGE SUBROUTINE CREATOR", D.P. Szetela, Nibble (Nov/Dec 80).

"TWO M/L SOUND EFFECTS PROGRAMS REVISITED", J.P. Davis, The Abacus][(Nov/Dec 80).

"ANIMATION WITH DATA ARRAYS", P. Connelly, Call A.P.P.L.E. (Dec 80).

"MULTIPLYING ON THE 6502", B.W. Boering, Micro (Dec 80).

"A BASIC TO MACHINE LANGUAGE INTERFACE", P. Rowe, Apple Orchard (Winter 80).

"A MACHINE LANGUAGE ADDRESS CALCULATOR", R. Lavallee, Apple Orchard (Winter 80).

"BINARY-TO-DECIMAL SHORTCUT (SMALL IS BEAUTIFUL)", S. Wozniak, Apple Orchard (Winter 80).

"POKE SALAD", T.A. Brown, Apple Orchard (Winter 80). An ampersand-invokable relocatable 150-byte machine language subroutine for displaying dollars and cents.

"SEARCHING STRING ARRAYS", G.B. Little, Micro (Jan 81). Relocatable machine language subroutine to search one dimensional string arrays.

"REAL VARIABLE STUDY", E.E. Goez, Call A.P.P.L.E. (Jan 81).

"APPLESOFT SUB-STRING SEARCH FUNCTION", L. Reynolds, Call A.P.P.L.E. (Jan 81).

"FAST GARBAGE COLLECTION", R. Wiggington, Call A.P.P.L.E. (Jan 81).

"AMPER-READER", A.G. Hill, Nibble (Jan/Feb 81).

"A GENERAL MOVE SUBROUTINE", B. Sander-Cederlof, Apple Assembly Line (Jan 81).

"A COMPUTED GOSUB FOR APPLESOFT", B. Sander-Cederlof, Apple Assembly Line (Jan 81).

"IN THE HEART OF APPLESOFT", C. Bongers, Micro (Feb 81).

"DOS INTERNALS: AN OVERVIEW", M. Pump, Call A.P.P.L.E. (Feb 81).

"APPLE NOISES AND OTHER SOUNDS", B. Sander-Cederlof, Apple Assembly Line (Feb 81).

"A STRING SWAPPER FOR APPLESOFT", B. Sander-Cederlof, Apple Assembly Line (Feb. 81).

"AMPER-SWITCH", B.E. Colley, Nibble (Feb/Mar 81).

"INPUTTING STRINGS WITH COMMAS", R.M. Mottola, Nibble (Feb/Mar 81).

"A BEAUTIFUL DUMP", R.H. Bernard, Apple Assembly Line (Mar 81).

"NOTES ON HI-RES GRAPHICS ROUTINES IN APPLESOFT", C.K. Mesztenyi, Apple Orchard (Spring 81).

"PASSING ARGUMENT VALUES TO MACHINE LANGUAGE SUBROUTINES IN APPLESOFT", C.K. Mesztenyi, Apple Orchard (Spring 81).

"RAPID BUBBLE SORT OF NUMERICAL ELEMENTS USING BASIC/ASL" , L.S. Reich, Micro (Mar 81).

"APPLE MEMORY MAPS, PART 2", P.A. Cook, Micro (May 81).

"APPLESOFT INTERNAL ENTRY POINTS", B. Sander-Cederlof, Apple Assembly Line (Apr 81).

"FAST INPUT STRING ROUTINE", B. Sander-Cederlof, Apple Assembly Line (Apr 81).

"SUBSTRING SEARCH FUNCTION FOR APPLESOFT", B. Sander-Cederlof, Apple Assembly Line (Apr 81).

"HIDING THINGS UNDER DOS", R. Hatcher, Apple Assembly Line (Apr 81). See corrections in June 81.

"APPLESOFT VARIABLE DUMP", S.D. Shram, Micro (May 81).

"HI-RES SCRN FUNCTION FOR APPLESOFT", B. Sander-Cederlof, Apple Assembly Line (May 81).

"TWO FANCY TONE GENERATORS", M. Kreigsman, Apple Assembly Line (Jun 81).

"MORE ABOUT MULTIPLYING ON THE 6502", B. Sander-Cederlof, Apple Assembly Line (Jun 81).

"HELLO-&", G. Teman, Nibble (May/Jun 81).

"APPENDING MACHINE LANGUAGE TO BASIC", M. Cross, Call A.P.P.L.E. (Jun 81).

"AMPERSEARCH FOR THE APPLE", A.G. Hill, Micro (Jun 81).

"BENEATH APPLE DOS", Quality Software, 1981.

"APPLE][MONITOR PEELED", Apple Computer, Inc., 1981.

"MACHINE LANGUAGE RANDOM NUMBER GENERATOR", B. Logan, Nibble (Jul/Aug 81).

"TRAPPING THE RESET KEY", G. Little, Nibble (Jul/Aug 81).

"COMMON ARRAY NAMES IN APPLESOFT][", S. Cochard, Micro (Aug 81). Contains ampersand-invokable machine language subroutine for interchanging numerical variables.

"STAND-ALONE RANDOM FUNCTION", B. Sander-Cederlof, Apple Assembly Line (Aug 81).

"FINDING APPLESOFT LINE NUMBERS", R.W. Potts, Apple Assembly Line (Aug 81).

"FIELD INPUT ROUTINE FOR APPLESOFT" R.W. Potts, Apple Assembly Line (Sep 81).

"CHRGET AND CHRGOT IN APPLESOFT", B. Sander-Cederlof, Apple Assembly Line (Sep 81).

"ALL ABOUT APPLESOFT", Call A.P.P.L.E. 1981. Contains numerous assembly language routines amoung the fourteen articles, most of which are likely to be of considerable interest to users of Routine Machine.

"AMPLIFYING APPLESOFT", D. Lingwood, All About Applesoft (1981).

"PRINT USING & FRIENDS", C. Peterson, All About Applesoft (1981).

"ULTIMATE INPUT-ANYTHING ROUTINE", P. Meyer, All About Applesoft (1981).

"WHAT'S WHERE IN THE APPLE?", W.F. Luebbert, Micro Ink, Inc., 1981.

"A SHORT APPLESOFT LINE FINDER ROUTINE", P. Meyer, Micro (Dec 81).

>> ROUTINE MACHINE: LIBRARY MODULES <<

The following pages detail each of the routines provided on the Routine Machine diskette. In addition, Ampersoft Program Library diskettes are available which contain additional routines for use with Routine Machine. Ask your dealer, or write Southwestern Data Systems for more information on these packages.

Each library entry describes both the function of the routine and the syntax expected when using it. Syntax is shown using the conventions defined in the Applesoft Reference Manual, pages 144 to 149. Although these abbreviations may seem a bit stiff at first, the goal was for accuracy and clarity in presentation.

Note that a number of routines introduce a new capability to Applesoft, namely, hexadecimal numbers. Most routines that allow hex numbers expect the number to be entered directly (not in string or numeric variable form). Syntax expressions involving hex numbers will always show these numbers as ´hexnum´. A hex number is defined as a dollar sign ($) followed by 1 to 4 hex digits.

One routine, XNUM.RM allows hex numbers within a string, and it that case, the string is required. The definition of a hex string for syntax expressions is ´hexstr´. A hex string must also begin with a dollar sign and be followed by 1 to 4 hex digits. All hex numbers must be in the range of $0 to $FFFF.

In addition to the formal syntax definition, several examples are usually given showing actual program lines using the given routine.

The length of each routine is also shown, to give you some idea of what trade offs are involved, if any, in implementing the routine. For example, PRINT USING.RM has a length of 261 bytes. This is considerably shorter than any alternative written in Applesoft, not to mention more efficient. In this case there is no trade off to speak of in implementing the routine. On the other hand, the HIRES ASCII.RM character set has a length of 1190 bytes, and should be considered when adding this to a program. You may find that this is sufficient to push the end of program point past the beginning of the Hi-Res page 1 display area of memory.

If you find your Hi-Res programs are getting so large as to conflict with the Hi-Res pages, you may wish to consider Ampersoft Program Library - Volume 3. This package is devoted to Chart Graphics and includes an automatic "splitter routine" that will wrap an Applesoft program around the Hi-Res pages to eliminate any memory conflicts that would otherwise have occurred. Ask your dealer for more information on this and other library diskettes.

For all routines, length should be noted to see what the addition is "costing" you. However, in most cases, routines are 100 to 200 bytes long, which is very comparable to the length of only one or two lines in an Applesoft program.

>> SWAP.RM <<

by Craig Peterson

FUNCTION: Swaps two Applesoft variables without requiring a third variable, and more importantly, without generating "garbage" which would have to be handled later.

LENGTH: 58 bytes ($3A)

SYNTAX: &"NAME",avar,avar or... &"NAME",svar,svar

SAMPLE: &"SWAP",A$,B$ or... &"SWAP",X,Y

HOW TO USE IT: This is most often used in sort routines and in general whenever you want to set one variable equal to another without generating additional "dead strings" in memory. If either variable does not exist, one will be created under that name, and given a null value. Thus, after the swap, the "new" variable will hold the old variable's data, and the "old" variable will be null.

LIMITATIONS: You cannot use variable expressions such as:

```
&"SWAP",A$,MID$(B$,1,1)
```

Both variable types must match; you could not use:

```
&"SWAP",A,A%
```

SAMPLE LISTING:

```
  1 CALL PEEK(175) + 256 * PEEK(176) - 46
 10 INPUT"STRING1, STRING2?";A$,B$
 20 PRINT A$,B$
 30 &"SWAP",A$,B$
 40 PRINT A$,B$: REM NOW SWAPPED
```

DEMONSTRATION PROGRAM: SWAP DEMO

>> PRINT USING.RM <<

by Craig Peterson

FUNCTION: Formats numeric data for screen, printer or disk file output.

LENGTH: 261 bytes ($105)

SYNTAX: &"NAME",edit string; number(s) to print
[;string suffix][;]
&"NAME",sexpr;aexpr [{,aexpr}]
[;sexpr][;]

Special Characters:

$	: fixed or floating dollar sign.
,	: fixed commas. Not printed when inappropriate.
0	: Number of places to round to when used AFTER the decimal point. Will act as a leading fill character when used BEFORE the decimal point.
*	: Fill character for unused digit positions. Often used on checks to mask leading blanks to amount.
space	: neutral position where a digit may be placed.
all others	: '/', ':', and all other characters will be used as non-replaceable characters around which the digits of the number will be filled. For example, &"PRINT","00/00/00",122581 would print as: 12/25/81

SAMPLE: &"PRINT","$, , .00";1234567.895;" TOTAL"
gives=> $1,234,567.90 TOTAL

Note that the length of the edit string (the no. of spaces between the quotes) fixes the length of the print field. All numbers printed will be right justified within this field.

And here's more:

```
NUMBER          EDIT STRING         PRINTOUT

123.571113      "       "              124
1491625.36      "             "      1491625
-1827.64125     "         .00"        -1827.64
11235813        "00000000000"  0011235813
-126.24         "00000"              -00126
1234567         "      "              #####
3.14159265      " 0.0000"                 3.1416
1.414213e-03    "0.000000000"             0.001414213
173205081       "    ,   ,    " 173,205,081
113.355         "     ,  $.00"          $113.36
2468.1012       "$    ,   .00"  $     2,468.10
1803.1763       "$        .00"  $      1803.18
1215.1492       "********"        ****1215
83886.08        "****,**$.00"     *$83,886.08
13.579          "TOTAL=  .00"     TOTAL=13.58
120744          "00/00/00"          12/07/44
630             "TIME = 0:00"     TIME = 6:30
241752148       "000-00-0000"     241-75-2148
```

HOW TO USE IT: PRINT USING.RM is an excellent and very compact way of formatting numbers which would be printed by an Applesoft program. Not only can monetary amounts be done with ease, but also things like dates and social security numbers are possible using the proper edit string field.

The edit string provides a picture of the field that the number will be placed into. It can be either a string literal, string variable, or string expression.

Numbers will be placed in the edit field from right to left. A decimal point in the field defines the number of places that the number will be rounded to. Any '0's, '$'s, '*'s or blanks in the field are POSITIONS where digits of the number may be placed. All other characters are non-replaceable.

Any commas in the field will be printed if embedded in the number. Commas to the left of the number will not be printed. Any '$'s in the field will be printed as positioned, unless pushed to the left by the number, thereby giving both fixed and floating dollar sign capabilities.

The number will be rounded to the decimal accuracy specified by the edit string field. Up to 9 significant digits can be printed. If the number is too large or the field too small, the entire field will be printed as '#'s.

If you wish to print more than one number expression using the same edit string field, just separate the numbers with commas. They will then be printed on the same line, tabbed as would have happened with the normal Applesoft PRINT statement.

When converting programs which used PRINT USING in this form:

```
10 $###,###.##
20 X = 50: PRINT USING 10
```

Routine Machine's PRINT USING would look like:

```
10 E$="    ,  $.00"
20 X = 50: &"PRINT USING",E$;X
```

If a semi-colon is used to terminate the numbers, then no carriage return will be printed, as occurs with the usual Applesoft PRINT statement.

LIMITATIONS: PRINT USING.RM will format numeric data only, i.e. a number must follow the edit string, NOT another string.

SAMPLE LISTING:

```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 INPUT"AMOUNT OF CHECK?";N
20 &"PRINT USING","$.00";N
```

DEMONSTRATION PROGRAM: PRINT USING DEMO

>> TEXT OUTPUT.RM <<

by Peter Meyer

FUNCTION: Prints text to screen or printer without word breaks at right margin.

LENGTH: 197 bytes ($C5)

```
SYNTAX: &"NAME",string [,width | ;]
        &"NAME",sexpr  [,aexpr | ;]

SAMPLE: &"TEXT",A$
        &"TEXT",A$+B$
        &"TEXT",A$,80
        &"TEXT",A$;
        &"TEXT",A$,80;
```

HOW TO USE IT: This routine is similar to the Applesoft PRINT statement, except that it is designed to eliminate the splitting of words at the ends of lines.

As in the standard PRINT statement, the presence of a semicolon at the end signifies that no carriage return is to be printed after the final character.

If the width parameter is not specified then it is assumed to be 40. Thus you can omit this parameter when outputting to the screen with the text window set normally. When sending text to a printer, the width should be set to the maximum number of columns supported (usually 80).

When text is being displayed on the screen, you may indent the first word (as at the start of a paragraph) by using the HTAB command to position the cursor appropriately before calling the routine. When printing multiple sentences, it is recommended that you define each string with two spaces following the period at the end. This will then properly format entire paragraphs to the screen. Note that leading spaces in any string are always eliminated.

LIMITATIONS: The text printed must be a string; numeric data is not allowed. You may however use Applesoft's STR$() function to convert any number to a string.

The width parameter must not be less than 20 or greater than 132, and must be at least as great as the length of the longest word in the text.

SAMPLE LISTING:

```
10  CALL PEEK(175) + PEEK(176) * 256 - 46: HOME
20  A$(1) = "ONCE UPON A TIME THERE WAS A DARK AND DEEP
    FOREST, FILLED WITH MYSTERIOUS PRESENCES, "
30  A$(2) = "LIKE ELVES, TROLLS, PIXIES, GIANTS AND
    ORCS.  "
40  A$(3) = "ONE DAY BO-PEEP CAME ALONG, LOOKING FOR
    HER SHEEP ... "
40  J = 1
50  HTAB J
60  FOR I = 1 TO 3
70  & "TEXT", A$(I);
80  NEXT: PRINT: PRINT
90  J = J+5:IF J < 30 THEN 50
```

DEMONSTRATION PROGRAM: STRING INPUT/TEXT OUTPUT DEMO

NOTE: To see how this routine may be used, in essentially the same way, to output a lot of text to the screen, the ARRAY SEARCH DEMO program may be studied.

>> STRING INPUT.RM <<

by Craig Peterson

FUNCTION: An alternative to the usual Applesoft INPUT statement, allowing commas and colons in input strings.

LENGTH: 41 bytes ($29)

SYNTAX: &"NAME",[prompt string;] string variable
&"NAME",[string;] svar

SAMPLE: &"INPUT","ENTER A STRING";A$(I)
&"INPUT",A$

HOW TO USE IT: When inputting strings which may (or may not) contain control characters, commas, quotation marks or colons, either directly from the keyboard, or from text files on disk.

LIMITATIONS: Numeric variables may not be input using this routine. ESCAPE characters/sequences will still be treated as editing sequences.

SAMPLE LISTING:

```
  1 CALL PEEK(175) + 256 * PEEK(176) - 46
 10 &"INPUT","ENTER A STRING: ";A$
 20 PRINT A$
 30 IF NOT (A$ = "END") THEN PRINT: GOTO 10
```

DEMO PROGRAM: STRING INPUT/TEXT OUTPUT DEMO

>> STRING SEARCH.RM <<

by Craig Peterson and Roger Wagner

FUNCTION: Finds a substring within another larger string.

LENGTH: 140 bytes ($8C)

SYNTAX: &"NAME","string to be searched","string to search for", position found at [,start position]
&"NAME",sexpr, sexpr, avar [,aexpr]

SAMPLE: &"SEARCH", "THIS IS A TEST", "THIS", P

&"SEARCH", "THIS IS A TEST", "TEST", P, N+5

HOW TO USE IT: This command is similar to the INSTR$ operator found in other BASICs. It is designed to find a given string within another string.

When using STRING SEARCH.RM the string to be searched (or its name) follows the command name. Immediately following that is the string variable, string literal or string expression for the substring which is to be searched for within the main string. After that is placed the variable in which the found position (if any) will be returned. If no occurrence is found, that variable will be set to "0".

Additionally you may place another parameter at the end of the command to specify the character position to begin the search at. If this parameter is omitted, then a starting position of "1" is assumed.

LIMITATIONS: The starting character position, if used, must have a value in the range of 1-255. String data (NOT NUMERIC) must be used for both main string and search string parameters.

SAMPLE LISTING:

```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 INPUT"STRING TO SEARCH:";I$
20 INPUT"STRING TO SEARCH FOR?";S$: N = 1
30 &"SEARCH",I$,S$,P
40 IF P=0 THEN PRINT"NOT FOUND":END
50 PRINT"FOUND AT POSITION ";P
60 N = P + 1: GOTO 30
```

DEMONSTRATION PROGRAM: STRING SEARCH DEMO

>> ARRAY SEARCH.RM <<

by Peter Meyer and Craig Peterson

FUNCTION: To search a one-dimensional string array for the occurrence of a specified search string, or for the occurrence of a string standing in a certain relation to the search string.

LENGTH: 456 bytes ($1C8)

SYNTAX: &"NAME",first element to search/ element # found
[,last element] ,string to search for,
[char posn] [,beg byte] [,search type]

&"NAME",array svar (avar) [TO array svar
(aexpr)], sexpr, [,avar] [,aexpr] [aexpr]

SAMPLE: &"ARYSRCH",A$(FE),KW$

=> Returns element of A$() in 'FE' at which string 'KW$' was found.

&"ARYSRCH",A$(FE) TO A$(LE),KW$,CP,BB,ST

=> Returns element of A$() in 'FE' at which string 'KW$' was found, when search was started at byte position 'BB' within each string. Search type is either 'full' or 'initial' depending on value (0-5) of 'ST'. On return CP = position in the string at which KW$ was found.

&"ARYSRCH",A$(FE),KW$,,BB

=> Search for KW$ starting at FE-th element. Examine each element beginning with the BB-th character and going through the rest of the string. Search to the end of the array, and don't bother to return the position at which search string found.

&"ARYSRCH",A$(FE),KW$,,,3

=> As above with the exception that no byte position is specified, and search type is to be type '3'.

HOW TO USE IT: ARRAY SEARCH.RM allows you to search quickly through a 1-dimensional string array, not only for identical matches, but also for 'relational' matches, that is, 'greater than' or 'less than' relationships between the search string and the strings in the array.

The simplest form of the invocation of this routine is as follows:

```
& "ARYSRCH", A$(FE), KW$
```

This may be translated as: Search through the array A$(), beginning with element A$(FE), looking for the first occurrence of KW$. (The array name, first element, and the keyword are all required in all forms of the module invocation.) On return, FE will be negative if KW$ was not found. Otherwise FE will be set to the index of the element in which KW$ was found.

Having found the element in which KW$ occurs, you can then display it, store it, or whatever. The routine returns when it finds an occurrence of KW$, and so does not in itself return ALL occurrences of KW$ in the array A$(). This can, however, be accomplished simply by repeated invocations of the routine. For example, the following line will display ALL occurrences of KW$ in an array B$ from the 1st element to the 100th (assuming that the length of B$ is at least 100):

```
200  FE = 1: & "ARYSRCH", B$(FE) TO B$(100), KW$:
     IF NOT (FE < 0) THEN PRINT B$(FE): IF NOT
     FE < 100 THEN FE = FE + 1: GOTO 200
```

If you wish to know the position in the string at which KW$ was found then add an extra variable to the parameter list, so:

```
&"ARYSRCH", A$(FE), KW$, CP
```

On return CP will be zero if KW$ was not found, otherwise it will be such that KW$ occurs at the CPth position in A$(FE) (remember that the value of FE has probably changed).

If you want to begin searching within each string from some position other than the first byte, include one more variable in the parameter list, so:

```
& ARYSRCH", A$(FE), KW$, CP, BB
```

BB is the Byte-to-Begin parameter, and must be in the range 1 <= BB <= 255.

So far all examples have been requests for a 'full' search, meaning that each string in the specified search range is examined from the BB-th character to the end of the string. Thus if we were searching for "CAT" with BB = 5 then the keyword would be found in an element such as "DOGS AND CATS" (and on return CP, if included in the parameter list, would be set to 10). (KW$ = "DOG" would not be found if BB = 5.)

But perhaps we wish to know if "CAT" occurs, not just somewhere in the string at or after the BB-th byte, but SPECIFICALLY AT the BB-th byte. This is a different TYPE of search, and so we must include yet another variable in the parameter list, so:

```
& "ARYSRCH", A$(FE), KW$, CP, BB, ST
```

ST is the Search Type, and can have any integer value in the range 0 - 5. ST = 0 means a full search, meaning (as explained above) that it will look through the entire string for the keyword. If ST > 0 then an 'initial' search will be done, which means that each string will be checked only as regards the substring which starts at the BB-th position (if BB = 1 then this means: the start of the string, hence 'initial search').

If ST > 0 then the routine will return when it finds a substring X$ at the BB-th position in some element in the array satisfying the following:

```
If ST = 1 then X$ = KW$
If ST = 2 then X$ < KW$
If ST = 3 then X$ <= KW$
If ST = 4 then X$ > KW$
If ST = 5 then X$ >= KW$
```

These concepts are illustrated further in the ARRAY SEARCH DEMO program.

ARRAY SEARCH.RM allows you to default on parameters in two ways. The first method is simply termination of the parameter list before specifying all variables. All parameters given must be in the correct order, but if you stop the list then the remaining variables are given their default values.

The second method is to create a null entry for a parameter by omitting the variable name where it would have appeared. This allows you to default on some parameters early in the list while still being able to give non-default values to parameters later in the list. For example, to default on all parameters except the search type one would use a module invocation such as:

```
'     & "ARYSRCH", B$(I), KW$, , , ST
```

The default values for the defaultable parameters are as follows:

```
LE = last element in the array
BB  =  1 (first character in each array element)
ST = 0 (full search)
```

LIMITATIONS: The relational searches mentioned above involve string comparisons only, e.g. "CAT" < "DOG". Searches which involve numerical comparisons, e.g.:

```
23 < 45 < 231
```

will function properly provided that the numbers are represented as strings left-justified with zeroes so that they have a common length, as in:

```
"0023" < "0045" < "0231"
```

The starting element must be specified as A$(FE), with FE defined appropriately. FE cannot be replaced by a number, since the routine returns with a value in FE.

SAMPLE LISTING:

```
 5  CALL PEEK(175) + PEEK(176)*256 - 46: TEXT:HOME
10 FOR I = 1 TO 10: FOR J = 1 TO 8
20 A$(I) = A$(I) + CHR$(48 + RND(1)*10): NEXT I
30 PRINT I; HTAB 5: PRINT A$(I): NEXT J: POKE 34,11
40 INPUT "DIGIT(S) TO FIND?   ";KW$: IF KW$ = ""
   THEN POKE 34,0: END
50 FE = 1: & "ARYSRCH", A$(FE), KW$, CP, 2:
   REM NOTE BB = 2
60 IF FE < 0 THEN PRINT "NOT FOUND";: GOTO 80
70 PRINT "FOUND";
80 HTAB 15: PRINT "FE = ";FE;"   CP = ";CP
90 PRINT: GOTO 40
```

DEMONSTRATION PROGRAM: ARRAY SEARCH DEMO

>> BUBBLE SORT.RM <<

by Peter Meyer

FUNCTION: Sorts the elements of a 1-dimensional string array, placing all empty strings at the end.

LENGTH: 250 bytes ($FA)

SYNTAX: &"NAME",A$(first) TO A$(last), [position of sort field [,length of sort field [, speaker flag]]]
&"NAME",A$(avar) TO A$(aexpr) [,aexpr [,aexpr[,aexpr]]]

SAMPLE: &"SORT",A$(FE) TO A$(LE)
&"SORT",A$(FE) TO A$(LE), PF
&"SORT",A$(FE) TO A$(99), PF, LF
&"SORT",A$(19) TO A$(LE), PF, LF, SPKR

HOW TO USE IT: This routine sorts strings according to the numerical order of their constituent hexadecimal bytes. This corresponds to the ASCII ordering (and thus to the alphabetical ordering) of characters as shown on page 94 of this manual. For example, if a string array contained the following eight strings then they would be sorted as shown:

CAT
D
DOGMA
RAT
RAT A TAT
RAT-A-TAT
RATATAT
RATS

If you have a string array B$ of length 100, then to sort the entire array simply use the command:

& "SORT", B$(0) TO B$(100)

You can sort a subrange of an array by defining FE and LE appropriately.

You might wish to sort the elements of an array on only a part of each string. For example, suppose you have an array containing strings such as:

842ROBERTS
090JONES
055SMITH
230BROWN

If you wanted to sort these strings simply on the name part then you could use the command:

```
& "SORT", A$(FE) TO A$(LE), 4
```

which means: Sort on only that part of each string beginning at the 4th character. If you wanted to sort on only the numeric part of these strings then you could use the command:

```
& "SORT", A$(FE) TO A$(LE), 1, 3
```

which means: Sort on the first three character only.

More generally, you can sort on a particular 'field' within a string by defining PF to be the position of the field and LF to be the length of the field. (See the SORT DEMO 2 for an illustration of this technique.)

The routine comes with optional sound effects in the form of clicking. In the case of sorts whose duration exceeds the average human boredom/anxiety coefficient, clicks are often entertaining and/or reassuring. The value of the final parameter, SPKR, determines the presence or absence of clicks. The routine will do its work silently if SPKR = 0.

You can default on the parameters as shown in the sample module invocations above. The default values are:

```
PF = 1     (first character in each string)
LF = 255   (sort on whole string)
SPKR = 1   (produces clicking)
```

Both elements in the sort range must be specified (unlike ARRAY SEARCH.RM), and the indices may be numerical variables or numbers (again unlike ARRAY SEARCH.RM, where the index of the first element must be a numerical variable).

LIMITATIONS: This is a string array sort, and will not normally work with arrays of numerical elements - but see Note (1) below.

The specification of the sort range must be such that FE <= LE. PF and LF must be > 0 and < 256, and such that PF + LF < 256.

NOTES: (1) Numbers can be sorted provided that they are represented as strings and left-justified with spaces or zeroes so that each string (or field within a string) has the same length. (See the SAMPLE LISTING below.)

(2) Although the Bubble Sort is slow in comparison with other sorts (and is insufferably slow in BASIC), it is surprisingly fast when done in machine language. BUBBLE SORT.RM's performance was measured using arrays of 10-character strings. The sorting times for arrays of various sizes were found to be as follows:

Number of elements in the array:	Sorting time in minutes and seconds:
125	0:02
250	0:06
500	0:23
1000	1:34
2000	6:17

These times can be confirmed using the SORT DEMO 1 program. The sorting time depends mainly on the size of the array, and very little on the size of the strings within the array. Doubling the array size has the effect of quadrupling the sorting time.

SAMPLE LISTING:

```
  1  CALL PEEK(175) + PEEK(176)*256 - 46
  5  FOR K = 1 TO 5: POKE 32,(K-1)*8: POKE 33,40-
     (K-1)*8:HOME:REM THIS HAS NOTHING TO DO
     WITH THE ACTUAL SORTING
 10  FOR I = 0 TO 9
 20  A$(I) = STR$(INT(RND(1)*1000)): REM RANDOM
     NUMBERS TO SORT
 30  A$(I) = RIGHT$("    "+A$(I),3): NEXT I:
     REM DON'T FORGET TO LEFT-JUSTIFY
 40  GOSUB 100: REM DISPLAY NUMBERS
 50  PRINT
 60  & "SORT", A$(0) TO A$(9): REM SORT 'EM!
 70  GOSUB 100: NEXT K:REM DISPLAY SORTED NUMBERS
 80  POKE 32,0: POKE 33,40:END
100  FOR I = 0 TO 9: PRINT A$(I): NEXT I: RETURN
```

DEMONSTRATION PROGRAMS: SORT DEMO 1

SORT DEMO 2

>> BEEP.RM <<

by Craig Peterson

FUNCTION: Generates a pure tone of a given pitch and duration. Can also be used to pause.

LENGTH: 56 bytes ($38)

SYNTAX: &"NAME" [,pitch [, duration]]
&"NAME" [,aexpr [, aexpr]]

SAMPLE: &"BEEP"
&"BEEP",P
&"BEEP",P,D
&"BEEP",P+5,D/2

HOW TO USE IT: The BEEP.RM routine can be used in a variety of ways. If no parameters are included after the name, then a tone having about the same pitch and duration of the normal Apple beep will be sounded.

If one parameter is included after the command name, it will change the pitch of the tone. This parameter can be any numeric constant, variable or expression, but must have a value from 0 to 255. Lower values cause higher pitches. A value of ´0´ produces no sound. If the pitch value is doubled, then a tone about one octave lower will be produced.

If a second parameter is given, the length of the tone can be changed. The length parameter can also be any numeric constant, variable or expression having a value between 0 and 255. This length value is determined in approximately 1/100´s of a second, so ´100´ would produce a tone of about one second in length. This allows tones with lengths ranging from 1/100 to over 2.5 seconds in length.

With pitch set to ´0´, the duration can be adjusted to produce silent waits of 1/100 to 2.5 seconds.

Be sure to run the BEEP DEMO to see what the musical possibilities of this routine are.

LIMITATIONS: Both pitch and duration values must be numeric variables in the range of 0-255.

SAMPLE LISTING:

```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 INPUT "PITCH, DURATION?";P,D
20 &"TONE",P,D
30 GOTO 10
```

Here is a table of the note values produced with different values for pitch:

NOTE	PITCH NO.	PITCH NO.	PITCH NO.
F SHARP	135	67	33
F	143	71	35
E	151	75	37
E FLAT	160	80	40
D	170	85	42
C SHARP	180	90	45
C	191	95	47
B	202	101	50
B FLAT	214	107	53
A	227	113	56
A FLAT	241	120	60
G	255	127	63

AND.. 31 (G)·

DEMONSTRATON PROGRAM: BEEP DEMO

>> SOUND EFFECTS <<

by Darrel Aldrich and David Lingwood

FUNCTION: Allows unusual sound effects in programs.

LENGTH: 28 bytes ($1C)

SYNTAX: &"NAME",pitch,shape
&"NAME",aexpr,aexpr

SAMPLE: &"SOUND",P,S
&"SOUND",10,125
&"SOUND",P+5,S/10

HOW TO USE IT: Both parameters must be provided when using this routine.

Pitch is a value in the traditional sense, with higher tones being produced by lower values. Pitch may be specified by a numeric constant, variable or expression with a value in the range from 1 to 255.

Shape is a value which will produce widely different results depending on its combination with pitch. It must also be specified with a numeric constant, variable or expression with a value in the range from 1 to 255.

You may wish to make note of combinations you find useful and/or interesting on the following page.

LIMITATIONS: Both parameters must be in the range of 1 to 255. Pure tones are difficult to produce using this routine.

SAMPLE LISTING:

```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 INPUT"PITCH,SHAPE?";P,S
20 &"SOUND",P,D
30 GOTO 10
```

DEMONSTRATION PROGRAM: SOUND EFFECTS DEMO

>> FIX LINK FIELDS.RM <<

by Peter Meyer

FUNCTION: To recalculate and establish the link fields in an Applesoft program.

LENGTH: 66 bytes ($42)

SYNTAX: & "NAME", avar

SAMPLE: & "FIX FIELDS", START

HOW TO USE IT: First make sure you understand what the link fields of an Applesoft program are (if in doubt, study Appendix B of this manual).

This routine is indispensable for all those times when you feel like moving an Applesoft program to another location in memory while the program itself is being executed. It is not difficult simply to copy an Applesoft program to another part of memory (this can be done easily using the MEMORY MOVE.RM routine on the Routine Machine diskette), but the program will not function at the new location unless the link fields are rectified.

Having moved an Applesoft program to a new location, invoke this routine with the command:

& "FIX FIELDS", START

where START is the new location of the program (that is, the address of the first link field, which is normally $800 = 2049).

Actually, it is unlikely that many programmers will find much use for this routine, since normally one does not move an Applesoft program around in memory. Nevertheless, there are some occasions where a routine such as this is required. For example, it is used in the BINARY FILE COPIER program on the Routine Machine diskette. In that program, if circumstances require it, the program makes a copy of itself in another part of memory and shortly thereafter, by a mysterious step, it is the copy, not the original program, that is being executed. Unfortunately an explanation of such esoteric matters falls outside the scope of this manual.

>> ERR.RM <<

by Apple Computer and Roger Wagner

FUNCTION: This fixes the stack pointer in preparation for continuing the operation of a running Applesoft program when RESUME will not be used. It will also optionally return the error code and line number of the error.

LENGTH: 63 bytes ($3F)

SYNTAX: &"NAME" [,error code [, line number]]
&"NAME" [,avar [, avar]]

SAMPLE: &"ERR"
&"ERR",EC
&"ERR",EC,EL

HOW TO USE IT: When the ONERR flag has been set within an Applesoft program, any errors encountered will go to the line number specified by the ONERR statement, at which point an error handling routine presumably exists. After the error has been handled, the programmer has the option of ending the program or resuming operation.

Normally, a RESUME statement is used to continue program operation. If RESUME is not used however, and the error occurred within a GOSUB or FOR-NEXT loop, problems will occur when the next RETURN or NEXT statement is encountered. This can be avoided by using the ERR.RM routine at the beginning of your error handling routine.

In addition, you may optionally include two other variable names after the command name. For the first numeric variable specified, the Applesoft or DOS error code will be returned in the variable given. A table of the possible error codes and their translations is given on the next page.

If the second variable is given, it will be returned with the line number on which the error occurred. This can also be rather useful. See the additional notes later in this section for an example of how this can be combined with the GOTO.RM routine to create a new variation on the RESUME function of Applesoft.

LIMITATIONS: No significant limitations.

SAMPLE LISTING:

```
    1 CALL PEEK(175) + 256 * PEEK(176) - 46
   10 ONERR GOTO 10000
   20 REM: ERR OCCURS HERE
10000 &"ERR",EC,EL
10010 PRINT"ERROR CODE:";EC
10020 PRINT"ON LINE #: ";EL
10030 POKE 216,0
10040 GOTO 20
```

>> DOS AND APPLESOFT ERROR CODES <<

ERR CODE	DESCRIPTION
0	NEXT WITHOUT FOR
1	LANGUAGE NOT AVAILABLE
2,3	RANGE ERROR
4	WRITE PROTECTED
5	END OF DATA
6	FILE NOT FOUND
7	VOLUME MISMATCH
8	I/O ERROR
9	DISK FULL
10	FILE LOCKED
11	SYNTAX ERROR
12	NO BUFFERS AVAILABLE
13	FILE TYPE MISMATCH
14	PROGRAM TOO LARGE
15	NOT DIRECT COMMAND
16	SYNTAX ERROR
22	RETURN WITHOUT GOSUB
42	OUT OF DATA
53	ILLEGAL QUANTITY
69	OVERFLOW
77	OUT OF MEMORY
90	UNDEFINED SUBSCRIPT
107	BAD SUBSCRIPT
120	REDIMENSIONED ARRAY
133	DIVISION BY ZERO
163	TYPE MISMATCH
176	STRING TOO LONG
191	FORMULA TOO COMPLEX
224	UNDEFINED FUNCTION
254	BAD RESPONSE TO INPUT STATEMENT
255	CONTROL-C INTERRUPT ATTEMPTED

A VARIATION ON RESUME:

Normally when RESUME is executed, the program resumes operation at exactly the same STATEMENT where the error occurred. This is sometimes inconvenient if what you would prefer is to have the entire LINE re-executed.

For example, in this program:

```
10 ONERR GOTO 100
20 N = N + 1
30 PRINT"AMOUNT OF CHECK #";N;":";:INPUT AMT(N)
40 GOTO 20

100 RESUME
```

If a string rather than a number is entered for AMT(N), a bad response to input error (#254) would be generated. When the RESUME was executed on line 100, the program would then continue on the INPUT AMT(N) statement on line 30 WITHOUT re-printing the prompt string. By combining ERR.RM with GOTO.RM, you can make the program resume operation with the entire line on which the error occurred.

```
10 ONERR GOTO 100
20 N = N + 1
30 PRINT"AMOUNT OF CHECK #";N;":";:INPUT AMT(N)
40 GOTO 20

100 &"ERR",EC,EL
110 IF EC=254 THEN &"GOTO",EL
120 RESUME
```

In this program, if a bad response to input error occurs, line 30 will be re-run from the beginning. See the section for the GOTO.RM routine for more details on that particular command.

DEMONSTRATION PROGRAM: ERR DEMO

>> ERR MSSG.RM <<

by Craig Peterson

FUNCTION: Prints the usual Applesoft or DOS error message from within a running program without halting program execution.

LENGTH: 150 bytes ($96)

SYNTAX: &"NAME"
&"NAME" [, error code]
&"NAME" [, aexpr]

SAMPLE: &"ERRMSSG" (will print error message for error code currently at location 222)
&"ERRMSSG",42 (will print OUT OF DATA)
&"ERRMSSG",EC (wIll print error message associated with error code EC)

HOW TO USE IT: Once you have set up the Applesoft ONERR GOTO statement, when an error occurs, it is always up to the programmer to print his own error messages. Quite often this means a lot of IF-THEN testing to see what message should be printed in the event of a standard error.

ERR MSSG.RM allows you to print any standard error message from within your program. To use it, simply follow the command name with the error code appropriate to the error that occurred. You may use the ERR.RM routine to retrieve the error code.

LIMITATIONS: The error code specified should be in the range of 0 to 255, and should correspond to an actual error message value. If values other than the ones specified in the chart under ERR.RM are given, partial messages may result.

No carriage return is printed after the error message, so print statements following the call will be printed immediately after the end of the previous message. This can be useful when adding text of your own to the standard message, but may cause difficulties if no carriage return is printed after the error message and a DOS command is then attempted (such as a PRINT D$;"CATALOG").

Note that messages for error codes 254 and 255 cannot be printed using this routine.

SAMPLE LISTING:

```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 ONERR GOTO 100
20 FOR I=1 TO 10
30 PRINT"VALUE FOR ITEM ";I;": ";:INPUT AMT(I)
40 NEXT I

100 &"ERR",EC
110 &"ERR MSSG",EC:PRINT
120 RESUME
```

DEMONSTRATION PROGRAM: ERR MSSG DEMO

>> GOTO.RM <<

by Roger Wagner

FUNCTION: Allows the equivalent of Applesoft's GOTO statement with the line number specified being given by a variable.

LENGTH: 43 bytes ($2B)

SYNTAX: &"NAME",line number
&"NAME",aexpr

SAMPLE: &"GOTO",LN
&"GOTO",100 * LN
&"GOTO",1000 + (X = 3.14) * 500

HOW TO USE IT: This can be very handy in menu routines where you want to go to a certain section of your program depending on a value entered by the user. Variable GOTO statements are also used to set up alternative ways of processing data depending on the results of an operation.

The line number to 'GOTO' can be specified by any numeric constant, variable or expression in the range of 0 to 65535.

For the third example, if LN can have values from 1 to 3, normal Applesoft would have been written:

```
10 IF LN=1 THEN GOTO 100
20 IF LN=2 THEN GOTO 200
30 IF LN=3 THEN GOTO 300

or...

10 ON LN GOTO 100,200,300
```

In the fourth example above, normal Applesoft would have to say:

```
10 IF X = 3.14 THEN 1500
20 GOTO 1000
```

LIMITATIONS: The line number must be specified by a numeric value, strings ARE NOT allowed. If the line number specified is not in the program, a UNDEFINED STATEMENT ERROR will be generated.

ZERO PAGE USAGE: None.

SAMPLE LISTING:

```
     1 CALL PEEK(175) + 256 * PEEK(176) - 46
    10 INPUT"LINE NUMBER TO 'GOTO'?";LN
    20 &"GOTO",LN

   100 PRINT"100":END
   200 PRINT"200":END
```

DEMONSTRATION PROGRAM: GOTO DEMO

>> GOSUB.RM <<

by Bob Sander-Cederlof

FUNCTION: Allows the equivalent of Applesoft's GOSUB statement with the line number specified being given by a variable.

LENGTH: 35 bytes ($23)

SYNTAX:
```
&"NAME",line number
&"NAME",aexpr
```

SAMPLE:
```
&"GOSUB",LN
&"GOSUB",100 * LN
&"GOSUB",1000 + (X = 3.14) * 500
```

HOW TO USE IT: This can be very handy in menu routines where you want to go to a certain section of your program depending on a value entered by the user. Variable GOSUB statements are also used to set up alternative ways of processing data depending on the results of an operation.

The line number to 'GOSUB' can be specified by any numeric constant, variable or expression in the range of 0 to 65535.

For the second sample line, if LN can have values from 1 to 3, normal Applesoft would have been written:

```
10 IF LN=1 THEN GOSUB 100
20 IF LN=2 THEN GOSUB 200
30 IF LN=3 THEN GOSUB 300

or...

10 ON LN GOSUB 100,200,300
```

In the third sample line above, normal Applesoft would have to say:

```
10 IF X = 3.14 THEN GOSUB 1500
20 GOSUB 1000
```

LIMITATIONS: The line number must be specified by a numeric value, strings ARE NOT allowed. If the line number specified is not in the program, a UNDEFINED STATEMENT ERROR will be generated.

ZERO PAGE USAGE: None.

SAMPLE LISTING:

```
     1 CALL PEEK(175) + 256 * PEEK(176) - 46
    10 INPUT"LINE NUMBER TO ´GOSUB´?";LN
    20 &"GOSUB",LN
    30 END
   100 PRINT"100":RETURN
   200 PRINT"200":RETURN
```

DEMONSTRATION PROGRAM: GOSUB DEMO

>> LINE DATA RESTORE.RM <<

by Roger Wagner

FUNCTION: Performs a similar function to Applesoft's RESTORE command, with the exception that the line number to which the DATA pointer is restored can be specified.

LENGTH: 49 bytes ($31)

SYNTAX: &"NAME",line number
&"NAME",aexpr

SAMPLE: &"RESTORE",1000
&"RESTORE",100 * X

HOW TO USE IT: Using LINE DATA RESTORE.RM it is possible to set up data structures within an Applesoft program where given blocks of line numbers contain distinct groups of information. When you want to access a given group, you no longer need to execute a "dummy" FOR-NEXT loop to read through all preceding entries in other DATA groups. Also, data can be added to other groups without affecting access to the group of interest.

To use this routine, simply follow the command name with a numeric constant, variable or expression whose value corresponds to a line number within your program. At that point, the DATA pointer will be RESTOREd to that line number, and all successive READ statements will operate respective to that starting point.

It should also be noted that strings within DATA statements take up less memory than actual array variables (no look up table is needed).

Note about when to use DATA vs. disk files: is data going to change?
y = disk, n = DATA
memory? DATA < ARRAYS

The line number specified must exist in the program, or a UNDEFINED STATEMENT ERROR will be generated. Also, DATA statements should be on, or follow after the line number specified, or an OUT OF DATA error will occur when a READ is attempted.

SAMPLE LISTING:

```
  1 CALL PEEK(175) + 256 * PEEK(176) - 46
 10 INPUT"DATA GROUP TO PRINT?";G
 20 &"RESTORE",G * 100
 30 FOR I=1 TO 5
 40 READ A$: PRINT A$
 50 NEXT I
 60 GOTO 10

100 DATA A,B,C,D,E
200 DATA 1,2,3,4,5
300 DATA A1,B2,C3,D4,E5
```

DEMONSTRATION PROGRAM: DATA RESTORE DEMO

>> DATA ELEMENT SELECT.RM <<

by Roger Wagner

FUNCTION: This advances the DATA pointer a given number of positions relative to its current position. This gives a random access-like aspect to DATA statements.

LENGTH: 129 bytes ($81)

SYNTAX: &"NAME",position
&"NAME",aexpr

SAMPLE: &"SELECT",X
&"SELECT",5 * N

HOW TO USE IT: This command can become a very powerful enhancement to Applesoft DATA statements depending on how it is implemented. To use it, simply follow the command name with the number of positions you would like to advance the DATA pointer. The position value may be given by a numeric constant, variable or expression.

When executed, the DATA pointer will be advanced the given number of elements from its current position. A value of "0" does not advance the pointer at all.

There are two main ways of using this routine. The first is for replacing the usual "dummy" FOR-NEXT loops which gobble unwanted DATA statement elements before locating the one you want. This can happen any time during a program when you want to advance the pointer to a new set of data. This operation is done much faster using this routine.

The second application is done by combining this routine with the standard RESTORE or the special LINE DATA RESTORE.RM module included with Routine Machine. By executing a RESTORE followed by the SELECT routine, a given element in a DATA group can be accessed very quickly. See the sample listing for an example of this.

LIMITATIONS: The value for the number of positions to advance must be a positive number. Also, attempts to advance the pointer more positions than there are DATA elements will generate an OUT OF DATA error.

SAMPLE LISTING #1:

```
  1 CALL PEEK(175) + 256 * PEEK(176) - 46
 10 INPUT "WHICH ELEMENT TO ACCESS?";E
 20 RESTORE: &"SELECT",E: READ A$
 30 PRINT A$
 40 GOTO 10

100 DATA 0,1,2,3,4,5
```

SAMPLE LISTING #2:

```
  1 CALL PEEK(175) + 256 * PEEK(176) - 46
 10 INPUT "WHICH MONTH NUMBER ? (1-12)";N
 20 &"RESTORE",200:&"SELECT",N:READ M$
 30 PRINT "MONTH NUMBER ";N;" IS ";M$
 40 GOTO 10

100 DATA DAYS,MONDAY,TUESDAY,WEDNESDAY,
         THURSDAY, FRIDAY, SATURDAY,SUNDAY

200 DATA MONTHS,JANUARY,FEBRUARY,MARCH,APRIL,
         MAY,JUNE,JULY,AUGUST,SEPTEMBER,
         OCTOBER,NOVEMBER,DECEMBER

300 DATA DEPARTMENTS,SALES,ACCOUNTING,SHIPPING,
         RECEIVING
```

NOTE: Each data list includes a zeroth element (example "MONTHS" in line 200). Because SELECT advances the data pointer 'N' positions PAST the first (or more accurately, the "current") entry, it is necessary to provide a "dummy" data element. This actually can be viewed as something of a "feature" in that it allows you to start each data list with an identifying string, as is done in SAMPLE LISTING #2.

SAMPLE LISTING #3:

```
  1 CALL PEEK(175) + 256 * PEEK(176) - 46
 10 INPUT "WHICH PART NUMBER?";PT$
 20 RESTORE
 30  READ  I$:IF I$ = "END" THEN PRINT "NOT FOUND":END
 40 IF I$ <> PT$ THEN &"SELECT",3:GOTO 30
 50 PRINT"PART NAME:";PN$:END
100 DATA 001A,GENERATOR,$29.95,25
110 DATA 002A,WIRE,$4.95,30
120 DATA 003A,BOLTS,$0.39,100
130 DATA END
```

NOTE: Use of "SELECT" here allows you to skip over data entries when scanning a table.

>> XNUM.RM <<

by Roger Wagner and Craig Peterson

FUNCTION: This will convert numbers between decimal and hexadecimal notations.

LENGTH: 214 bytes ($D6)

SYNTAX: &"NAME",hex string variable,decimal real variable
&"NAME",decimal real variable,hex string variable
&"NAME",sexpr|aexpr, avar|svar

SAMPLE:
```
&"XNUM","$200",N     (N will come back = 512)
&"XNUM",512,H$       (H$ will come back = $0200)
&"XNUM",H$,N
&"XNUM",N,H$
```

HOW TO USE IT: How you use XNUM.RM will depend on whether you are converting decimal to hexadecimal, or vice versa.

To convert decimal to hex, simply follow the command name with any numeric constant, variable, or expression. Then follow that with the string variable into which you wish the hex equivalent string placed. The result will then be placed in that string, at which point it may be printed, or passed to another routine.

To convert hex to decimal, follow the command name with a string variable or expression containing the hex number to be converted. The string must begin with a dollar sign ($) and be one to four digits long (not including the dollar sign). Then follow that with a real or integer variable into which the decimal equivalent will be placed. After the call, the variable will be equal to the decimal value for the hex string specified. If a real variable is used, then results can be in the range of 0 to 65535. If an integer variable is used, then results greater than 32767 will be expressed in the negative form.

LIMITATIONS: The hex string must be a string variable, whose contents are a legitimate hex number, beginning with a dollar sign. Numbers from $0000 to $FFFF can be converted. When converting decimal to hex, the decimal value must be in the range of -65535 to 65535.

SAMPLE LISTING:

```
  1 CALL PEEK(175) + 256 * PEEK(176) - 46
 10 INPUT"NUMBER TO CONVERT?";N$
 20 IF LEFT$(N$,1) = "$" THEN 100
 30 N = VAL(N$)
 40 &"XNUM",N,H$
 50 PRINT N;" = ";H$
 60 END

100 &"XNUM",N$,N
120 PRINT N$;" = ";N
130 END
```

>> MEMORY MOVE.RM <<

by Roger Wagner

FUNCTION: Moves (or more accurately, copies) a block of memory from one location to another.

LENGTH: 248 bytes ($F8)

SYNTAX: &"NAME",beginning of source, end of source,
beginning of destination
&"NAME",aexpr|hexnum,aexpr|hexnum,aexpr|hexnum

SAMPLE: &"MOVE",4096, 8192, 16381
&"MOVE",$2000,$3FFF,$4000
&"MOVE",AD,AD+4096,$4000

HOW TO USE IT: To use this routine, you must know the addresses of the beginning and end of the range of memory you wish to move, and the address of the beginning of the destination block.

If specifying the address(es) in decimal, you may use any numeric constant, variable, or expression in the range of -65535 to 65535.

If specifying the address(es) in hex, you must enter the number as a hex literal. Strings are not allowed, although XNUM.RM may be used to convert existing hex strings to decimal if needed, prior to calling this routine.

Decimal and hex notations may be freely mixed.

Memory blocks may be moved either up or down in relation to their starting position, and the beginning of the destination range may lie within the source block itself with no ill effects.

LIMITATIONS: Strings may not be used to specify hex numbers. They must be hex literals (no quotes around the number). In addition, all values whether decimal or hex, must be in the range -65535 to 65535 or $0 to $FFFF. The value for the end of the source block must also be greater than the address for the beginning of the source block.

SAMPLE LISTING:

```
  1 CALL PEEK(175) + 256 * PEEK(176) - 46
 10 HOME:HGR2
 20 HCOLOR=3: HPLOT 0,0 TO 150,150
 30 HGR
 40 INPUT"PRESS <RETURN> TO MOVE PG2 TO PG1";I$
 50 &"MOVE",$4000,$3FFF,$2000
 60 END
```

DEMONSTRATON PROGAM: MOVE DEMO

>> RESTORE AMPERSAND.RM <<

by Craig Peterson

FUNCTION: This will restore the ampersand vector to its original value, as it was before the Applesoft program was run.

LENGTH: 18 bytes ($12)

SYNTAX: &"NAME"

SAMPLE: &"AMP"

HOW TO USE IT: If you are using any utilities which use the ampersand vector, running one of your own programs with Routine Machine modules in it will re-write that vector. When your program ends, the previously operational utility (if there was one) will no longer respond to the ampersand in the immediate mode.

To correct this, an optional procedure has been provided by way of this routine. When your program first connects the ampersand vector via the hook-up on line #1, the Interface Routine stores the original value of the ampersand vector.

When you are going to END the Applesoft program, simply invoke the RESTORE AMPERSAND.RM routine and the ampersand will be restored to its earlier function.

LIMITATIONS: None per se, although a fair degree of care is required in its use. If you exit a program via an error, control-C, RESET or any other manner other than the controlled (and expected) exit you previously set up, it is likely the RESTORE AMPERSAND.RM routine will not be called. This would then leave the ampersand still pointing to the Interface Routine. If you were then to re-run your program, the Interface Routine would again save the current status of the ampersand, which would now point to itself. In other words, any previous ampersand related utilities would now be permanently disconnected.

NOTE: If you think you understand how the module RESTORE AMPERSAND.RM is meant to be used, then it is good practice to use it in all programs containing Routine Machine modules. It MUST, however, be the LAST module invoked before your program ceases execution, since the invocation of this routine disables ALL appended modules.

If you invoke RESTORE AMPERSAND.RM, and then try to invoke another module, the result will depend on the pre-RUN value of the ampersand vector, but a likely result would be a SYNTAX ERROR.

SAMPLE LISTING:

```
  1 CALL PEEK(175) + 256 * PEEK(176) - 46

 10 REM USUAL R.M PROGRAM HERE

100 &"AMP":END
```

>> PTR READ.RM <<

by Roger Wagner and Craig Peterson

FUNCTION: This will read any two byte pointer in memory, and return the decimal value in a numeric variable.

LENGTH: 156 bytes ($9C)

SYNTAX: &"NAME",address,variable
&"NAME",aexpr|hexnum,avar

SAMPLE: &"PTRD",175,N
&"PTRD",AD,X
&"PTRD",$73,N

HOW TO USE IT: This is most useful when trying to determine the contents of the many two byte pointers that Applesoft uses to keep track of its own memory usage. To determine where one of these byte pairs is currently pointing, follow the command name with the address of the first byte in the pair.

The address may be specified in either decimal or hex notations. If decimal is chosen, follow the command name with a numeric constant, variable or expression.

If the address is to be specified in hex, simply use the hex value, beginning with a dollar sign ($) and having one to four hex digits.

After the specified address, place a real or integer variable into which the returned address is to be placed. Real variables will return with values in the range of 0 to 65535. If an integer variable is used, values over 32767 will be expressed in their negative forms. If a hex address is desired, XNUM.RM may be used to convert decimal addresses to hex if desired.

LIMITATIONS: If hex addressing is used, the address must be specified within a string. All addresses must be in the range of -65535 to 65535 or $0 to $FFFF.

SAMPLE LISTING:

```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 &"PTRD",$AF,N
20 PRINT"END OF PROGRAM: ";N
30 &"XNUM",N,H$
40 PRINT"(HEX:           ";H$;")"
```

>> PTR WRITE.RM <<

by Roger Wagner

FUNCTION: Sets any two byte pointer in memory to the value or address specified.

LENGTH: 150 bytes ($96)

SYNTAX: &"NAME",addr of pointer, value to put there
&"NAME",aexpr|hexnum,aexpr|hexnum

SAMPLE: &"PTRWRT",113,32768
&"PTRWRT",$73,$1000
&"PTRWRT",$AF,V

HOW TO USE IT: Applesoft uses many two byte pairs as pointers to various important locations in memory. It is often useful to change these pointers to indicate new locations.

To write data to a byte pair, simply follow the command name with the address of the byte pair, and then the address to which the pointer is to be set.

If the address is specified in decimal, it may be any numeric constant, variable or expression in the range of -65535 to 65535.

If the address is specified in hex, it must be a series of hex digits beginning with a dollar sign and in the range of $0 to $FFFF.

LIMITATIONS: All values must be in the range of -65535 to 65535 or $0 to $FFFF.

SAMPLE LISTING:

```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 PRINT CHR$(4);"BLOAD SHAPE.ASCII,A$4000"
20 &"PTRWRT",$E8,$4000: REM SET UP SHAPE TABLE
30 HGR: HCOLOR = 3: ROT = 0: SCALE = 1
40 DRAW 1 AT 100,100
```

>> SHAPE GOBBLER / SHAPE VIEWER <<

by Roger Wagner

SHAPE GOBBLER is a utility provided to convert existing Applesoft Shape Tables into usable Routine Machine library modules. Ordinarily, to use a shape table, one would have to go through a rather complicated procedure of loading and protecting the table in memory, setting pointers, etc. Needless to say, this makes the use of shape tables a bit inconvenient.

SHAPE GOBBLER solves this problem by processing raw shape tables into a simple ampersand callable routine. Once the table has been converted, you simply load the library module like you would any other library routine. When the routine is called, the shape table will then be automatically installed with all shape parameters (such as SCALE, ROTATION, etc.) set to their common defaults. From that point on, you may use all the normal Applesoft Shape Table commands such as DRAW, XDRAW, ROT, SCALE, etc. with no concern to the technical details of their implementation.

TO USE SHAPE GOBBLER:

Place the Routine Machine diskette in your drive and type in:

RUN SHAPE GOBBLER

The program will then prompt you to insert a diskette containing a known shape table. For purposes of example, you may now put the Routine Machine demo disk in the drive. When it asks for the shape table name, press RETURN alone to catalog the diskette. In the second half of the listing you should see a file called SHAPE.ASCII. Enter this name and press RETURN.

The program will then prompt you to insert the diskette you want the completed library module saved on, and to specify the name to save the file under. You can give the name TEST.RM for this example. The disk will then come on for a moment. When it is completed, you will have a usable shape table module which can be easily used with Routine Machine.

See the entry on HIRES.ASCII (and other shape tables) for details on how to use shape tables within your programs.

IMPORTANT: PLEASE NOTE THE DISTINCTION BETWEEN ´RAW´ SHAPE TABLES AND CONVERTED LIBRARY MODULES. SHAPE TABLES ARE DIFFICULT TO USE DIRECTLY. LIBRARY MODULES, ON THE OTHER HAND, ARE EASILY INSTALLED AND CALLED WITH A COMMAND NAME.

ONCE CONVERTED TO A MODULE, YOU CANNOT RUN SHAPE GOBBLER A SECOND TIME ON THE BINARY FILE. IT IS SUGGESTED YOU USE SOME SORT OF NAMING CONVENTION TO PREVENT POSSIBLE CONFUSION. THE ONE USED ON THE ROUTINE MACHINE DISKETTE IS TO USE THE PREFIX ´SHAPE.´ FOR RAW SHAPE TABLES AND THE SUFFIX ´.RM´ FOR COMPLETED LIBRARY MODULES.

>> SHAPE TABLES IN GENERAL <<

Shape tables are a powerful potential feature of Applesoft, but are rarely implemented because of two main drawbacks.

The first drawback is the difficulty of creating them in the first place. Although the Applesoft Reference Manual provides technical information as to the details of shape tables, they are for the most part difficult to create, at least manually.

This is most easily solved by purchasing one of the several programs commercially available for creating shape tables. Southwestern Data Systems is currently preparing such a utility to generate shape tables. Depending on when you are reading this section, it might already be available. Check with your local Apple Dealer or call Southwestern Data Systems for more information on this.

Even without creating your own shape tables, a number of tables are already available at little or no cost through local Apple user groups. If such a group exists in your area you may wish to inquire as to the availability of existing shape tables. These can then be converted using SHAPE GOBBLER for use via Routine Machine in your own programs.

The second problem with shape tables is installing them in a running Applesoft program. It has already been mentioned that this is ordinarily a rather difficult task, best accomplished by more experienced programmers.

With Routine Machine and SHAPE GOBBLER this is no longer a problem. The converted library modules are easily put in any program in exactly the same manner as any other library module.

Once called, the shape table is installed and all normal shape table related commands work just as described in the manual.

Note that The Printographer, a full-featured Hi-Res printing utility from SDS, can be used to easily print Hi-Res screens and is fully compatible with Routine Machine.

>> USING CONVERTED SHAPE TABLES <<

Once the table is converted, consider it just like a normal library module. Use these guidelines for using them:

FUNCTION: Invoking a shape table module connects that particular shape table to any successive Applesoft shape table commands such as DRAW, ROT, SCALE, etc.

SYNTAX: &"NAME",number of shapes
&"NAME",avar

EXAMPLE: &"SHAPE",N

HOW TO USE IT: Simply follow the command name with a numeric variable. This variable will be filled with the value for the number of shapes in the table.

When the module is called, the associated shape table will be connected to the appropriate Applesoft pointers, and the SCALE and ROTATION parameters set to 1 and 0, respectively.

The hardest part about explaining this command is that it is almost too simple!

LIMITATIONS: Virtually none. As long as you have converted a legitimate shape table using SHAPE GOBBLER everything should work just fine!

SAMPLE LISTING:

```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 HGR:HCOLOR=3
20 &"SHAPE",N
30 VTAB 22:PRINT"THERE ARE ";N;" SHAPES IN THE TABLE"
40 DRAW 1 AT 140,80
```

>> SHAPE TABLE VIEWER <<

by Roger Wagner

This utility is provided to allow you to examine raw shape tables. This can be useful, especially in cases where you might not even be sure if a binary file is in fact a shape table.

NOTE: SHAPE TABLE VIEWER WILL ONLY VIEW RAW SHAPE TABLES. IT WILL NOT SUCCESSFULLY DISPLAY CONVERTED LIBRARY MODULES.

To use the program, insert the Routine Machine diskette into your drive and type in: RUN SHAPE TABLE VIEWER.

The program will then present a brief explanation of its function, and then prompt you to insert a diskette containing a raw shape table into the drive, and press a key.

The diskette will then be cataloged, at which point you can enter the name of the shape table file you wish to display. For this example, you can enter the name SHAPE.ASCII.

When the table has been loaded, the program will then present four separate displays. The first display is done with SCALE = 1 and ROT (rotation) = 0. This corresponds to the most common display method for a table.

The next two displays set ROT = 16 and ROT = 48. These correspond to rotations of 90 degrees to the right and left, primarily so that you can see if the shapes look okay in these forms. For character sets such as SHAPE.ASCII this is important when labeling graphs, etc.

The last display is with SCALE = 2. It is my personal opinion that shape tables rarely display well with SCALE equal to anything but 1, and even rotations other than 0 should be seldom used. This display is provided however, so you can see just how well the table does stand enlargement. To make things look at least a little better, the figures are actually drawn twice with a slight offset to create a more "solid" figure. You can delete line number 151 of the SHAPE TABLE VIEWER program if you do not want the overdraw done.

This program does not save any files to disk, or in any way alter shape tables viewed.

>> HIRES ASCII.RM <<

by Stephen L. Billard

FUNCTION: This is a shape table of 95 ASCII characters used for printing text on the Hi-Res screen.

LENGTH: 1190 bytes ($4A6)

SYNTAX: &"NAME",number of shapes (always returns '95')
&"NAME",avar

SAMPLE: &"ASC",N

HOW TO USE IT: By invoking this module, the ASCII shape table is automatically connected. By DRAWing the appropriate shape number you can put any ASCII character on the screen.

By doing a FOR-NEXT loop on a string, it is possible to print a line of text to the screen. See SAMPLE LISTING #2 and the HIRES ASCII DEMO on the Routine Machine demo disk for examples of how to do this.

LIMITATIONS: Attempts to draw shapes close enough to the edge of the screen so that the character goes off-screen may result in some distortion. Same limitations as the Applesoft DRAW, etc. commands.

SAMPLE LISTINGS:

#1:
```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 HGR:HCOLOR=3
20 &"ASC",N
30 VTAB 22:PRINT"LETTER TO PRINT?";:GETA$:PRINT A$
40 C=ASC(A$)-32: IF C THEN DRAW C AT X,20
50 X = X + 6: GOTO 30
```

#2 (PRINTS A LINE OF TEXT):

```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 HGR:HCOLOR=3
20 &"ASC",N
30 VTAB 22:INPUT"TEXT TO PRINT?";I$
40 X=20:Y=20
50 FOR I=1 TO LEN(I$)
60 C = ASC(MID$(I$,I,1)) - 32
70 IF C THEN DRAW C AT X,Y
80 X = X + 6: NEXT I
90 REM CHANGE 80 TO Y=Y+8:NEXT I TO PRINT VERTICALLY
```

REMOVE RM before Running Program

SEE Pg 14

>> TURTLE GRAPHICS.RM / TURTLE GRAPHICS+.RM <<

by Michael Freeman

FUNCTION: Provides a set of Hi-Res graphics drawing routines based on the concept of "turtle graphics".

LENGTH: 612 bytes ($264) basic version
988 bytes ($3DC) "+" version

SYNTAX: &"NAME",command character [,parameter(s)]
&"NAME",character [,aexpr(s)]

The following provides information on each of the various commands available in TURTLE GRAPHICS.RM and TURTLE GRAPHICS+.RM.

GENERAL INFORMATION:

Turtle Graphics is based on the concept of an imaginary "turtle" which can be directed in various motions across the screen. Attached to our critter's tail is an equally imaginary pen. If the pen is "down" while he moves, a trail is left behind tracing his path. If the pen is "up", no trail is left and the turtle simply moves to a new position.

Drawing is usually initiated by starting the turtle out at a given position on the screen, facing a given direction, with the pen down. From there the turtle is directed to either "move" or "turn". Moves are given by a number of screen units, turns are specified in degrees. Turns can either be relative to the current direction, or an absolute angle, with zero degrees being defined as the turtle facing the right edge of the screen.

The advantages of Turtle Graphics over X,Y coordinate graphics are best utilized when doing shape related graphics. Whether to use Turtle Graphics or the usual HPLOT X,Y method is best judged by the individual programmer according to the application.

SPECIFIC COMMANDS:

Note that all commands begin with a command function letter, followed by a parameter list (except for 'INIT'). Where a value is specified by 'aexpr', this signifies the use of a numeric constant, variable or expression.

INIT &"TG",I

Places the turtle at 140,96 in direction zero degrees (facing right) with the pen "down". This command MUST be given before any of the other Turtle Graphics commands.

```
SET     &"TG",S,horiz,vert,angle
        &"TG",S,aexpr,aexpr,aexpr
        &"TG",S,X,Y,D
```

Places the turtle at X,Y in direction D (degrees).

```
PEN     &"TG",P,pen status
        &"TG",P,aexpr
        &"TG",P,P
```

If aexpr is non-zero, then the pen is "down" (plots). If aexpr is zero, the pen is "up" (does not plot).

```
TURN    &"TG",T,relative angle
        &"TG",T,aexpr
        &"TG",T,A
```

Turns the turtle relative to its current direction. Positive values correspond to a clockwise turn, negative to counterclockwise.

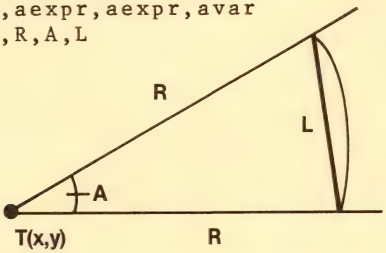
```
        &"TG",T,@A
        &"TG",T,@absolute angle
        &"TG",T,@aexpr
```

Turns the turtle to face an absolute angle (in degrees) given by aexpr.

```
MOVE      &"TG",M,distance
          &"TG",M,aexpr
          &"TG",M,S
```

Moves the turtle in its current direction 'S' units. If the pen is down, a trace in the current HCOLOR will be left. If the pen is up, no trace will be left.

```
CHORD     &"TG",C,radius,angle,length
          &"TG",C,aexpr,aexpr,avar
          &"TG",C,R,A,L
```



Given the radius 'R' And the angle in degrees 'A', this returns in the variable 'L' the length of the chord.

```
FIND      &"TG",F,horiz,vert,direction [,pen status]
          &"TG",F,avar,avar,avar, [,avar]
          &"TG",F,X,Y,D [,P]
```

Returns the current turtle position in terms of the X and Y coordinates, and the direction the turtle is facing (in degrees). An optional fourth variable may be included, into which will be placed a 1 or a 0 depending on the status of the pen. '1' signifies pen "down", '0' corresponds to pen "up". This has no effect on the current position or status of the turtle.

```
ARC       &"TG",A,radius,angle
          &"TG",A,aexpr,aexpr
          &"TG",A,R,A
```

Draws an arc of radius 'R' for an angle of 'A' degrees. Arc is started in direction the turtle is facing and proceeds clockwise for positive values of 'R' (Radius), and counter clockwise for negative Radius values. NOTE THAT THIS OPTION IS AVAILABLE ON THE '+' VERSION ONLY.

LIMITATIONS: Although normal Hi-Res graphics may be used in conjunction with the turtle routines, be aware that there is some interaction. Whenever the turtle is moved, the internal Hi-Res "cursor" is updated. Thus, the following commands:

```
&"TG",S,140,96,0: HPLOT TO 0,0
```

would draw a line from the center of the screen to the upper left-hand corner. Using this feature, any HPLOT TO command may use the current turtle position as a starting point. However, the converse is not true. You MAY NOT do a HPLOT to a position and then expect the turtle to go from there.

To summarize: Turtle move commands DO affect subsequent HPLOT commands. HPLOT commands DO NOT affect the turtle.

If an attempt is made to draw off the screen, an ILLEGAL QUANTITY error will be generated. Use SET or INIT to reposition the turtle back onto the screen. Any attempt to MOVE back may produce unpredictable results.

Values for X and Y coordinates must be within the normal permissable range for Applesoft Hi-Res graphics. (X in the range 0 to 279, Y in the range 0 to 159 (mixed screen) or 191 (full screen)).

Angle measurements may be greater than + or - 360 degrees, but will wrap-around. That is to say that specifying an angle of 370 degrees will have exactly the same result as specifying an angle of 10 degrees.

Two different versions of TURTLE GRAPHICS are included on the Routine Machine diskette. Although the ´plus´ version offers the additional command ´ARC´, it is somewhat longer. If memory is a problem, and you do not need the ARC function, then simply use the smaller version of the package, TURTLE GRAPHICS.RM.

SAMPLE LISTINGS:

#1: BOX

```
 1 CALL PEEK(175) + 256 * PEEK(176)-46
10 HGR: HCOLOR = 3
20 &"TG",I
30 FOR I = 1 TO 4
40 &"TG",M,50: &"TG",T,90
50 NEXT I
```

#2: POLYGONS

```
 1 CALL PEEK(175) + 256 * PEEK(176)-46
10 HGR: HCOLOR = 3
20 &"TG",I
30 FOR S = 3 TO 15
40 A = 360/S: &"TG",S,140,10,0
50 FOR J = 1 TO S
60 &"TG",M,30: &"TG",T,A
70 NEXT J
80 NEXT S
```

#3: DESIGN

```
 1 CALL PEEK(175) + 256 * PEEK(176)-46
10 HGR: HCOLOR = 3
20 &"TG",I
30 FOR D = 0 TO 360-10 STEP 10
40 &"TG",S,140,96,D
50 &"TG",M,60: &"TG",T,90: &"TG",M,60
60 &"TG",T,90: &"TG",M,60
70 NEXT D
```

#4: RAINBOW

```
 1 CALL PEEK(175) + 256 * PEEK(176)-46
10 HGR
20 &"TG",I
30 FOR R = 20 TO 70
40 HCOLOR = INT(R/10)-(R<50)
50 &"TG",S,140-R,100,-90
60 &"TG",A,R,180
70 NEXT R
```

DEMONSTRATON PROGRAM: TURTLE GRAPHICS DEMO

>> BLOAD.RM <<

by Steve Cochard

FUNCTION: BLOADs binary files approximately four times faster than normal.

LENGTH: 597 bytes ($255)

SYNTAX: &"NAME",filename [,address]
&"NAME",sexpr [,aexpr]

SAMPLE: &"BLOAD","HIRES PICTURE"
&"BLOAD",F$
&"BLOAD",F$,AD
&"BLOAD",F$,8192

HOW TO USE IT: Simply follow the command name with the name of the file to be loaded. Any string literal, variable or expression may be used to specify the file name.

If you wish to specify a load address, use any numeric constant, variable or expression after the file name.

LIMITATIONS: The optional load address must be specified by a numeric variable. Hex numbers or string variables may not be used. (Use the XNUM.RM routine if it is necessary to use hex addresses.) There is no provision for specifying slot, drive or volume. The last-accessed drive will thus always be selected.

SAMPLE LISTING:

```
 1 CALL PEEK(175) + 256 * PEEK(176) - 46
10 HGR
20 &"BLOAD","HIRES PICTURE",8192
```

DEMONSTRATION PROGRAM: BLOAD/BINADR DEMO

>> BINADR.RM <<

by Steve Cochard

FUNCTION: Returns the length and load address of any binary file on a disk, WITHOUT loading the file.

LENGTH: 443 bytes ($1BB)

SYNTAX: &"NAME",filename ,length [,load address]
&"NAME",sexpr, avar [,avar]

SAMPLE: &"BINADR","HIRES PICTURE",L,A
&"BINADR",F$,L

HOW TO USE IT: Simply follow the command name with the name of the file to be read. Any string literal, variable or expression may be used to specify the file name.

Then put the variable which you wish to have set equal to the length after the file name, separated by a comma.

If you wish to determine the load address, use another numeric variable, again separated from the length variable by a comma.

LIMITATIONS: The optional load address must be specified by a numeric variable. String variables will generate a TYPE MISMATCH error. There is no provision for specifying slot, drive or volume. The last-accessed drive will thus always be selected.

SAMPLE LISTING:

```
10 CALL PEEK(175) + 256 * PEEK(176) - 46
20 &"BINADR","HIRES PICTURE",L,A
30 PRINT"LENGTH: ";L
40 PRINT"ADDRESS: ";A
```

DEMONSTRATION PROGRAM: BLOAD/BINADR DEMO

>> RESET NORM.RM <<

by Roger Wagner

FUNCTION: This sets the RESET vector back to its normal condition, so that pressing RESET will put the user in direct command mode with an Applesoft prompt.

LENGTH: 16 bytes ($10)

SYNTAX: &"NAME"

SAMPLE: &"RESET NORM"

HOW TO USE IT: This is used primarily to clear other RESET conditions set up by RESET ONERR.RM, RESET RUN.RM, or RESET BOOT.RM. See the descriptions of those routines for details on their function.

RESET NORM.RM is used most frequently just before ending the program and returning the user to the normal operating system. (Note that RESET NORM.RM must be used before RESTORE AMPERSAND.RM, if the latter routine is also used before ending the program.)

LIMITATIONS: This assumes that the usual RESET vector bytes (at $3F2-3F4) of BF 9D 38 are the desired bytes. These are the usual values for DOS 3.3 on a 48K machine.

SAMPLE LISTING:

```
    1 CALL PEEK(175) + 256 * PEEK(176) - 46
   10 &"RESET BOOT" : REM USE RESET BOOT.RM
   20 REM
   30 REM YOUR PROGRAM HERE...
   40 REM
 1000 &"RESET NORM" : REM BACK TO NORMAL...
 1010 END
```

DEMONSTRATION PROGRAM: RESET DEMO

>> RESET ONERR.RM <<

by Roger Wagner and Peter Meyer

FUNCTION: This sets the RESET vector so as to generate an Applesoft error code when RESET is pressed. If an ONERR GOTO statement is in effect, control will then pass to the error-handling routine as in the case of a normal error occurrence.

LENGTH: 42 bytes ($2A)

SYNTAX: &"NAME",error code value
&"NAME",aexpr

SAMPLE: &"RESET ERR",99
&"RESET ERR",X

HOW TO USE IT: This is used to disable the usual RESET function and to define what error code will be generated when RESET is pressed. The value may be specified by any numeric constant, variable or expression in the range of 0 to 255. See the section on ERR.RM to avoid conflict with other already defined error codes.

LIMITATIONS: If an ONERR statement is not active, then RESET ONERR.RM will cause the program to halt and give the normal Applesoft prompt, as any other error would. Depending on the error code specified though, strange error messages may result. RESET ONERR.RM should therefore always be used in conjunction with the usual ONERR GOTO error trapping routines.

SAMPLE LISTING:

```
  1 CALL PEEK(175) + 256 * PEEK(176) - 46
 10 ONERR GOTO 100
 20 &"RESET ERR",99

 40 REM YOUR PROGRAM HERE...

100 &"ERR",EC
110  IF EC = 99 THEN PRINT"DON'T PRESS RESET!": RESUME
120 &"ERR MSSG", EC: PRINT
130 POKE 216,0: &"RESET NORM": END
```

>> RESET RUN.RM <<

by Roger Wagner and Peter Meyer

FUNCTION: This sets the RESET vector so as to re-run the current Applesoft program when RESET is pressed.

LENGTH: 24 bytes ($18)

SYNTAX: &"NAME"

SAMPLE: &"RESET RUN"

HOW TO USE IT: This is used to set up the RESET vector so as to re-run any program currently in memory. There are no additional parameters required.

LIMITATIONS: As with all routines which tamper with the RESET vector, RESET NORM.RM should be used to restore RESET to normal operation before exiting the program.

SAMPLE LISTING:

```
  1 CALL PEEK(175) + 256 * PEEK(176) - 46
 10 &"RESET RUN"
 20 REM
 30 REM YOUR PROGRAM HERE...
 40 REM

100 &"RESET NORM"
110 END
```

Free Sector Count Demo

03/04/83

```
LIST

10  CALL  PEEK (175) + 256 *  PEEK (176) - 46
20  TEXT : HOME : PRINT "Which drive: ";: GET Z$: PRINT : IF Z$ = "E" TH
EN  VTAB 3: PRINT "Bye...": VTAB 6: END
30 DR =  VAL (Z$): & "FSC",DR;FS
40  VTAB 1: PRINT "Drive #";DR;" has ";: INVERSE : PRINT FS;: NORMAL : P
RINT " sectors still available"
50  VTAB 6: PRINT "<Cr> to continue: ";: GET Z$: PRINT : GOTO 20

]
```

>> RESET BOOT.RM <<

by Roger Wagner

FUNCTION: This sets the RESET vector so as to re-boot DOS when RESET is pressed.

LENGTH: 6 bytes ($6)

SYNTAX: &"NAME"

SAMPLE: &"RESET BOOT"

HOW TO USE IT: This is used to set up the RESET vector so as to re-boot DOS if RESET is pressed. There are no additional parameters required.

LIMITATIONS: None (just use RESET NORM.RM before exiting the program).

SAMPLE LISTING:

```
  1 CALL PEEK(175) + 256 * PEEK(176) - 46
 10 &"RESET BOOT"
 20 REM
 30 REM YOUR PROGRAM HERE...
 40 REM

100 &"RESET NORM"
110 END
```

>> FREE SECTOR COUNT <<

by Peter Meyer

FUNCTION: Allows determination during program execution of the number of free sectors on a disk in a selected drive.

LENGTH: 116 bytes ($74)

SYNTAX: & "NAME"; avar
& "NAME", avar; avar

SAMPLE: & "FSC"; FS
& "FSC", DR; FS

CALL (768) 2; FS
& "FSC", 2; FS

HOW TO USE IT: If you are about to write a file to disk and you are unsure whether there is sufficient space on the disk then this module can be invoked to provide relevant information. On return, FS = the number of free sectors on the disk. You can specify the drive desired by including the 'DR' parameter as shown. If this is omitted then the last-accessed drive will be selected.

LIMITATIONS: The variable following the name must be a real variable, not an integer variable.

SAMPLE LISTING:

```
 5 CALL PEEK(175) + 256 * PEEK(176) - 46
10 & "FSC"; FS
20 PRINT "YOUR DISK HAS "FS" FREE SECTORS"
```

NOTES: If a disk I/O ERROR is encountered when the routine tries to access the disk then on return the value of FS will be negative.

This routine may also be used in direct command mode, as follows: BLOAD FREE SECTOR COUNT.RM then enter:

```
CALL 768;N: PRINT N
```

The number of free sectors on the disk will then be displayed on the screen.

Note that a semi-colon is used in the parameter list before the parameter FS. This illustrates a conventional usage of the semi-colon to separate the input variables from the output variables in a parameter list. This convention should be followed only in parameter lists in which the semi-colon is not also used in its other conventional meaning: to denote the suppression of a carriage return in a PRINT instruction.

DEMONSTRATION PROGRAMS

The reverse side of the Routine Machine diskette contains over twenty programs which demonstrate the use of the foregoing Routine Machine modules. This side of the diskette is not protected in any way, and a copy may be made using ordinary copy programs such as COPYA on the Apple DOS 3.3 Master Diskette.

Many of these programs output a considerable quantity of text to the screen. This is usually done using the TEXT OUTPUT.RM routine. The ARRAY SEARCH DEMO, or (for a simpler example) the FREE SECTOR COUNT DEMO, may be studied to see how this is done.

Special note should be taken of the program called BINARY FILE COPIER. This program demonstrates the combined use of SIXTEEN Routine Machine modules, ALL taken from the Routine Machine diskette. To find out which ones, BRUN ROUTINE MACHINE, turn to the reverse side of the disk, LOAD BINARY FILE COPIER, enter the Routine Machine with a CALL 2051 and get a module report. Then search the program for ampersand statements to find out just where and how these modules are invoked. Note the combined use of RESET ONERR.RM and RESET NORM.RM, and how ERR MSSG.RM makes use of information provided by ERR.RM. Note also the use of RESTORE AMPERSAND.RM as the very last module invoked before the program ceases execution.

BINARY FILE COPIER is a demo program with a difference: It is also a useful utility in itself. Copy the program onto your own work diskettes and use it whenever you wish to copy a binary file from a disk in one drive to a disk in another.

REMEMBER: Please don't hesitate to drop us a note telling us of your impressions of the Routine Machine and in what applications you are using it. We would especially appreciate any comments you may have as to the points you like best, or areas that may do with some improvement.

>> USER NOTES <<

This section is provided as a space to make notes for yourself regarding Routine Machine, and may also prove useful as a place to put descriptions of routines you may write yourself, or come across in magazines, etc.

As this section and your experience grows, please don't hesitate to drop us a note telling us of your impressions of Routine Machine, and what applications you're developing for it.

We especially appreciate any comments you may have as to the points you like best, or areas that may stand some improvement.

ROUTINE MACHINE MODULES - QUICK REFERENCE LIST

NOTE: This list is provided solely to refresh your memory as to the exact syntax of any given routine. It is assumed you have already read the primary reference section in the manual and are familiar with the routine.

SWAP.RM: (pg 103)

```
&"SWAP",avar,avar
&"SWAP",X,Y  or &"SWAP",X$,Y$
```

PRINT USING.RM: (pg 104)

```
&"PRINT",sexpr;aexpr [{,aexpr}] [;sexpr] [;]
&"PRINT","  $.00",2.65*1.05
&"PRINT",EDIT$,NUM1,NUM2;SUFFIX$;
```

TEXT OUTPUT.RM: (pg 107)

```
&"TEXT",sexpr [,aexpr] [;]
&"TEXT",A$ or &"TEXT",A$,80;
```

STRING INPUT.RM: (pg 109)

```
&"INPUT",[sexpr;] svar
&"INPUT",A$ or &"INPUT","PROMPT";A$
```

STRING SEARCH.RM: (pg 110)

```
&"SEARCH",sexpr,sexpr,avar [,aexpr]
&"SEARCH",STRNG$,SRCH$,FOUND
&"SEARCH",STRNG$,SRCH$,F,START
```

ARRAY SEARCH.RM: (pg 111)

```
&"ARYSRCH",array svar (avar) [TO array svar
(aexpr)] ,sexpr, [,avar] [,aexpr] [,aexpr]
&"ARYSRCH",A$(FE),SRCH$
&"ARYSRCH",A$(FE) TO A$(LE),SRCH$,CHAR,BYT,TYP
&"ARYSRCH",A$(FE),SRCH$,,,3
```

BUBBLE SORT.RM: (pg 115)

```
&"SORT",array svar (avar) TO array svar (avar)
        [,aexpr [,aexpr [,aexpr]]]
&"SORT",A$(FE) TO A$(LE)
&"SORT",A$(FE) TO A$(LE),PSN,LNGTH,SPKR
```

BEEP.RM: (pg 118)

```
&"BEEP" [,aexpr [,aexpr]]
&"BEEP" or &"BEEP",PTCH or &"BEEP",PTCH,DRTN
```

SOUND EFFECTS.RM: (pg 120)

```
&"SOUND",aexpr,aexpr
&"SOUND",PITCH,SHAPE
```

```
FIX LINK FIELDS.RM: (pg 121)
        &"LINKFIX",aexpr
        &"LINKFIX",STARTADDR

ERR.RM: (pg 122)
        &"ERR" [,avar [,avar]]
        &"ERR" or &"ERR",CODE or &"ERR",CODE,LINE

ERR MSSG.RM: (pg 125)
        &"MSSG" [,aexpr]
        &"MSSG" or &"MSSG",CODE

GOTO.RM: (pg 127)
        &"GOTO",aexpr
        &"GOTO",LINENUM

GOSUB.RM: (pg 129)
        &"GOSUB",aexpr
        &"GOSUB",LINENUM

LINE DATA RESTORE.RM: (pg 131)
        &"RESTORE",aexpr
        &"RESTORE",LINENUM

DATA ELEMENT SELECT.RM: (pg 133)
        &"SELECT",aexpr
        &"SELECT",PSN

XNUM.RM: (pg 135)
        &"XNUM",sexpr|aexpr, avar|svar
        &"XNUM",DEC,HEX$ or &"XNUM",HEX$,DEC

MEMORY MOVE.RM: (pg 137)
        &"MOVE",aexpr|hexnum,aexpr|hexnum,aexpr|hexnum
        &"MOVE",$2000,$3FFF,$4000
        &"MOVE",AD,AD+4096,$4000
        &"MOVE",$2000,4096,8192

RESTORE AMPERSAND.RM: (pg 139)
        &"AMP"

PTR READ.RM: (pg 141)
        &"PTRD",aexpr|hexnum,avar
        &"PTRD",ADDR,NUM or &"PTRD",$73,NUM

PTR WRITE.RM: (pg 142)
        &"PTRWRT",aexpr|hexnum,aexpr|hexnum
        &"PTRWRT",ADDR,NUM or "PTRWRT",$73,$380
        &"PTRWRT",ADDR,$380 or "PTRWRT",$73,NUM
```

```
HIRES ASCII.RM: (pg 149)
        &"ASC",avar
        &"ASC",NUM      {always returns '95')

TURTLE GRAPHICS.RM / TURTLE GRAPHICS+.RM: (pg 150)
        &"TG",character [,aexpr(s)]

Commands:

 INIT:   &"TG",I          {MUST BE USED FIRST}

 SET:    &"TG",S,aexpr,aexpr,aexpr
         &"TG",S,X,Y,DIR

 PEN:    &"TG",P,aexpr
         &"TG",P,UPDOWN (UPDOWN = 0 or 1)

 TURN:   &"TG",T,aexpr
         &"TG",T,ANGLE  (makes relative turn)

         &"TG",T,@aexpr
         &"TG",T,@ANGLE (turns to absolute angle)

 MOVE:   &"TG",M,aexpr
         &"TG",M,DIST

 CHORD:  &"TG",C,aexpr,aexpr,avar
         &"TG",C,RADIUS,DIR,LNGTH

 FIND:   &"TG",F,avar,avar,avar [,avar]
         &"TG",F,X,Y,DIR or &"TG",F,X,Y,DIR,PEN

 ARC:    (available in '+' version only)
         &"TG",A,aexpr,aexpr
         &"TG",A,RADIUS,ANGLE


BLOAD.RM: (pg 155)
        &"BLOAD",sexpr [,aexpr]
        &"BLOAD",NAME$ or &"BLOAD",NAME$,ADDRESS

BINADR.RM: (pg 156)
        &"BINADR",sexpr,avar [,avar]
        &"BINADR",NAME$,ADDR
        &"BINADR",NAME$,ADDR,LNGTH

RESET NORM.RM: (pg 157)
        &"RESET NORM"

RESET ONERR.RM: (pg 158)
        &"RESET ERR",aexpr
        &"RESET ERR",ERRCODE
```

RESET RUN.RM: (pg 159)
        &"RESET RUN"

RESET BOOT.RM: (pg 160)
        &"RESET BOOT"

FREE SECTOR COUNT.RM: (pg 161)
        &"FREE";avar or &"FREE",avar; avar
        &"FREE";FREE or &"FREE",DRIVE;FREE

>> POSSIBLE ERRORS... <<

SYNTAX ERROR....

1) Check syntax for the routine you're using.

2) Make sure the ampersand hook-up line (usually line number 1) has been installed and executed prior to the routine being invoked.

SYSTEM HANGS WHEN MODULE IS INVOKED...

1) Make sure the ampersand hook-up line (usually line number 1) has been installed and executed prior to the routine being invoked.

2) If the routine being called involves moving blocks of memory, make sure that the parameters specified are appropriate. Also make sure that you are not overwriting critical areas of the Apple's memory such as DOS, zero page, etc.

UNDEFINED FUNCTION ERROR...

1) Check to make sure you are using the same name to invoke a module as was used when first appending the module. You can use Option #6 to review the names you gave modules as they were appended.

FILE NOT FOUND (WHEN APPENDING A MODULE)...

1) Make sure you have included the '.RM' in the name of the module you wish to append.

2) Make sure the routine to be appended is on The Routine Machine diskette, an Ampersoft Program Library diskette, or a legitimate back up of these products.

Routine Machine & ProDos

1) Can't use any disk access routines
Eg Catalog Read etc.

2) In order to use CONVERT, you must have LOADED AND SAVED your Applesoft program while Routine Machine was NOT present. (RM usually moves your Applesoft program to $2601 - see page 38) Also DOS 3.3 always resets the link fields where as ProDos doesn't always do this.

To AVOID this conflict, after APPENDING with RM, BRUN REMOVE ROUTINE MACHINE, Then SAVE your program AND use CONVERT TO move it to ProDos.

Note: telltale sign is only Anyerrand Setup Line shows and Rest of Listing is GARBAGE

LOADER PROGRAM
TO LOAD BASIC PROGRAM
JUST ABOVE Hi-RES Pg 1
TO USE Hi RES GRAPHICS

```
10 POKE 104, 64
20 POKE 16384, 0
30 PRINT CHR$(4)"
   RUN FILENAME"
```

① 104 is BEG prgm pt
② APPLESOFT NEEDS 0
   BEG BYTE OF 1 PROGRAMS
③ RUNS desired program

SAMPLE LISTING:

```
    1 CALL PEEK(175) + 256 * PEEK(176) - 46
   10 ONERR GOTO 10000
   20 REM: ERR OCCURS HERE

10000 &"ERR",EC,EL
10010 PRINT"ERROR CODE:";EC
10020 PRINT"ON LINE #: ";EL
10030 POKE 216,0
10040 GOTO 20
```

>> DOS AND APPLESOFT ERROR CODES <<

ERR CODE	DESCRIPTION
0	NEXT WITHOUT FOR
1	LANGUAGE NOT AVAILABLE
2,3	RANGE ERROR
4	WRITE PROTECTED
5	END OF DATA
6	FILE NOT FOUND
7	VOLUME MISMATCH
8	I/O ERROR
9	DISK FULL
10	FILE LOCKED
11	SYNTAX ERROR
12	NO BUFFERS AVAILABLE
13	FILE TYPE MISMATCH
14	PROGRAM TOO LARGE
15	NOT DIRECT COMMAND
16	SYNTAX ERROR
22	RETURN WITHOUT GOSUB
42	OUT OF DATA
53	ILLEGAL QUANTITY
69	OVERFLOW
77	OUT OF MEMORY
90	UNDEFINED SUBSCRIPT
107	BAD SUBSCRIPT
120	REDIMENSIONED ARRAY
133	DIVISION BY ZERO
163	TYPE MISMATCH
176	STRING TOO LONG
191	FORMULA TOO COMPLEX
224	UNDEFINED FUNCTION
254	BAD RESPONSE TO INPUT STATEMENT
255	CONTROL-C INTERRUPT ATTEMPTED

255	CONTROL-C INTERRUPT ATTEMPTED
254	BAD RESPONSE TO INPUT STATEMENT
253	UNDEFINED FUNCTION
252	FORMULA TOO COMPLEX
150	STRING TOO LONG
140	TAPE MISMATCH
133	DIVISION BY ZERO
130	REDIMENSIONED ARRAY
101	BAD SUBSCRIPT
90	UNDEFINED SUBSCRIPT
77	OUT OF MEMORY
69	OVERFLOW
53	ILLEGAL QUANTITY
52	OUT OF DATA
51	RETURN WITHOUT GOSUB
16	SYNTAX ERROR
12	NOT DIRECT COMMAND
14	PROGRAM TOO LARGE
13	FILE TYPE MISMATCH
15	NO BUFFERS AVAILABLE
11	SYNTAX ERROR
10	FILE LOCKED
9	DISK FULL
8	I\O ERROR
7	VOLUME MISMATCH
6	FILE NOT FOUND
5	END OF DATA
4	WRITE PROTECTED
3	RANGE ERROR
1	LANGUAGE NOT AVAILABLE
0	NEXT WITHOUT FOR

ERR CODE DESCRIPTION

>> DOS AND APPLESOFT ERROR CODES <<

10000 GOSUB 30
10020 POKE 216,0
10030 PRINT ERR ON LINE ";PEEK
10040 PRINT"ERROR CODE:";DEC
10000 PRINT"ERR";DEC
30 REM ERR CODES HERE
40 ONERR GOTO 10000

Poke-16297,0 Hi Res

-16302,0 Full Page

Page 1 -16300,0 Pg 2 = -16299

-16304,0 Graphics

SDS

SIGMA 9 COMPUTER
REFERENCE MANUAL

VULCAN BINDERS
VINCENT ALA
35178